PACKAGE COHESION METRIC FOR OBJECT ORIENTED SYSTEMS

BY

SYED MANZOOR HUSSAIN

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION & COMPUTER SCIENCE

DECEMBER 2005

King Fahd University of Petroleum & Minerals
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Syed Manzoor Hussain** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

Dr. Jarallah AlGhamdi (Chairman)

Dr. Krishna Rao (Member)

Dr. Moataz Ahmed (Member)

Department Chairman
(Dr.Kanaan Faisal)

Dean of Graduate Studies
(Dr.Mohammad Abdul Aziz Al-Ohali)

Date 1-8-2006

Dedicated to My Parents, brother and sisters

Acknowledgments

All thanks are due Allah first and foremost for his countless blessings. Acknowledgement is due to KFUPM and the ICS Department for supporting this research.

I was honored and privileged to have Dr. Jaralla AlGhamdi as my advisor. His continual support and constructive criticism is highly appreciated and I feel deeply indebted to him for shaping this research with his extensive knowledge and experience. He was of constant support throughout the thesis and never did he make me feel uncomfortable in the problems I faced as a result of my limited knowledge. I am thankful to him for bearing with patience my every mistake, correcting it and encouraging at every step. I also wish to thank my thesis committee members, Dr. Moataz Ahmed for his constant guidance and developing new ideas which were detrimental to the achievement of this thesis and Dr. Krishna Rao for his help, support, and suggestions.

My thankful admiration goes to all my colleagues, especially Adnan Shahab for helping me whenever I faced problems in the thesis. Many of my ideas would have remained shallow and abstract but for the insightful discussions and questions at Software Metrics{ XE "Software Metrics" } Research Group (SMRG). My thanks go to all members of the SMRG and to all my friends, specially, Muqtader, Imran, Atif who have helped me through my thick and thin, also I extend my thanks to my seniors Noman bhai , Ameen bhai, Amjad bhai, Khaleel bhai, Ashfaq bhai, Ahsan bhai, Jaweed bhai and all the colleagues at KFUPM who provided wholesome companionship throughout my studies at KFUPM.

Above all I thank my family for all their support and love throughout my MS studies, without whose support it would have been impossible to continue my studies away from

home. Finally I thank KFUPM for the research facilities it provided which helped me achieve this.

Contents

TABLE OF CONTENTS

List of Figures

List of Tables

THESIS ABSTRACT

NAME: SYED MANZOOR HUSSAIN

TITLE: PACKAGE COHESION METRIC FOR OBJECT ORIENTED SYSTEMS

MAJOR FIELD: COMPUTER SCIENCE

DATE OF DEGREE: DECEMBER 2005

Software Metrics{ XE "Software Metrics" } can help a great deal to understand measure, analyze, control and improve software product and process{ XE "process" } attributes. Given the importance of object-oriented development techniques, one specific area where this has occurred is cohesion measurement in object-oriented systems. But the research on cohesion has only been limited at the class level, whereas the cohesion at the package level is also of utmost importance. Designing cohesive packages means creating packages that offer coarse-grained, yet much focused behaviors. In this thesis we present new package cohesion metrics that we have developed and implemented and also we will discuss the only other package cohesion metric that we have found in the literature, the component cohesion metric by Giancarlo et al, which we have also implemented using the connection criteria that we have come up with in our research. The results of the two metrics are then discussed.

ملخص الرسالة

الإسم : سيد منظور حسين

العنوان: التوافق على مستوى السلال - Packages - في البرمجيات المبنية على الأشياء

التخصص: علوم الحاسب

تاريخ الرسالة: ديسمبر 2005

لمقاييس البرمجيات دور كبير في قياس وتحليل منتجات وعمليات البرمجية مما يمكن تحديثها بشكل أفضل والتحكم بجودة هذه العمليات والمنتجات . ولما لتقنية البرمجة المبنية على الأشياء - Object-Oriented - من أهمية كبيرة في مجال تطوير البرامج. احد المقاييس المهمة لهذي التقنية هي قياس التوافق في الـ Object-Oriented. وقد اقتصرت الأبحاث في السابق على مستوى اللبنة– Class - في البرمجة المبنية على الأشياء. لذا وجدنا أن قياس التوافق على مستوى السلة - (Package Level) وهو مستوى أعلى من مستوى اللبنات حيث تُجمع عدة لبنات في سلة واحدة - سيكون مفيداً وغير مسبوق. إن تجميع لبنات متناسقة ومتوافقة في سلة واحدة يعني تناغم هذه اللبنات مما يزيد في سهولة فهمها وصيانتها.ولقد أثمر البحث في هذه الرسالة ثلاث مقاييس جديدة لدرجة التوافق على مستوى السلال. كما تم اختبار هذه المقاييس على عدة أنظمة وقمنا بعرض نتائج هذي الاختبارات وتحليلها ومناقشتها.

Chapter 1

Introduction

Software Metrics{ XE "Software Metrics" } can help to understand, measure, analyze, control, and improve software product and process{ XE "process" } attributes. Object-oriented technology is one of the most widely used paradigms for developing software systems because many researchers assert that OO practice assures good quality software. Through the years, many software attributes have been identified that have relation, in one way or the other, with the quality of the artifact being produced during the software process. Such attributes include: size, complexity, coupling, and cohesion. The Unified Modeling Language{ XE "Unified Modeling Language" } (UML{ XE "Unified Modeling Language (UML)" }) is widely used for expressing design artifacts in Object Oriented Software Development{ XE "Object Oriented Software Development(OOSD)" } (OOSD). The increasing importance being placed on software measurement has led to an increased amount of research developing new software measures. Given the importance of object-oriented development techniques, one specific area where this has occurred is cohesion measurement in object-oriented systems. Cohesion is a measure of quality indicating the degree to which some entity performs its intended purpose. Systems exhibiting high degrees of cohesion consist of elements that are focused on performing individual tasks, where each task contributes to the overall purpose of the system. Systems with lower degrees of cohesion have elements that perform a mish mash of functionality, where the purpose of each individual element is not well defined. When designing systems, we strive to achieve a high degree of cohesion, designing elements

that focus on performing a single task. But the research on cohesion has only been limited at the class level, whereas the cohesion at the package level is also of utmost importance. Designing cohesive packages means creating packages that offer coarse-grained, yet very focused, behaviors. Creating highly cohesive packages offers some significant, though often overseen advantages. For instance, packages are independently deployable. A package independent of any other package can be deployed as a reusable unit. Classes, on the other hand, can only be deployed with their containing package. Section 1.1 discusses the concept of object-oriented cohesion. Section 1.2 presents a brief introduction to the concept of package cohesion. Section 1.3 presents the UML. And Sections 1.4 and 1.5 present the motivation behind this research and the main contributions of the work, respectively. Finally section 1.6 shows the organization of the thesis.

1.1 Cohesion

Cohesion is an internal software attribute which depicts how well connected the components of a software module are. This can be determined by knowing the extent to which the individual components of a module are required to perform the same task [1]. In a highly cohesive module all the components performance are tailored towards the requirement of a single function. On the contrary, a low cohesive module has some elements that have little relationships with others, which is an indication that the module may provide several unrelated functions [2]. If a module is highly cohesive then it is easy to develop and maintain because it does not have much dependence on the components of other modules as such it is less error-prone.

Attributes such as coupling equally serve as quality indicators; coupling and cohesion are terms used to define module interconnectedness. Coupling is a measure of how strongly one element is connected to, have knowledge of, or relies on other elements [3].While cohesion addresses intra-module connectedness, coupling addresses inter-module connectedness. In general, coupling should be minimized while cohesion should be maximized [3] [4]. In object-oriented paradigm, however, coupling should not be completely minimized because some level of dependence is required for instance dependence due to inheritance is required.

1.2 Types of Cohesion

In the order form worst to best the different types of cohesion are as follows:

- *Coincidental:* Little or no constructive relationship among the elements of the module.
- *Logical:* Module performs a set of related functions, one of which is selected via function parameter when calling the module.
- *Temporal:* Elements are grouped into a module because they are all processed Within the same limited time period
- *Procedural*: Associates processing elements on the basis of their procedural or algorithmic relationships
- *Communication:* Operations of a module all operate upon the same input data set and/or produce the same output data.

- *Sequential:* Sequential association the type in which the output data from one processing element serve as input data for the next processing element.

- *Functional:* If the operations of a module can be collectively described as a single specific function in a coherent way, the module has functional cohesion.

This is depicted in the figure below:

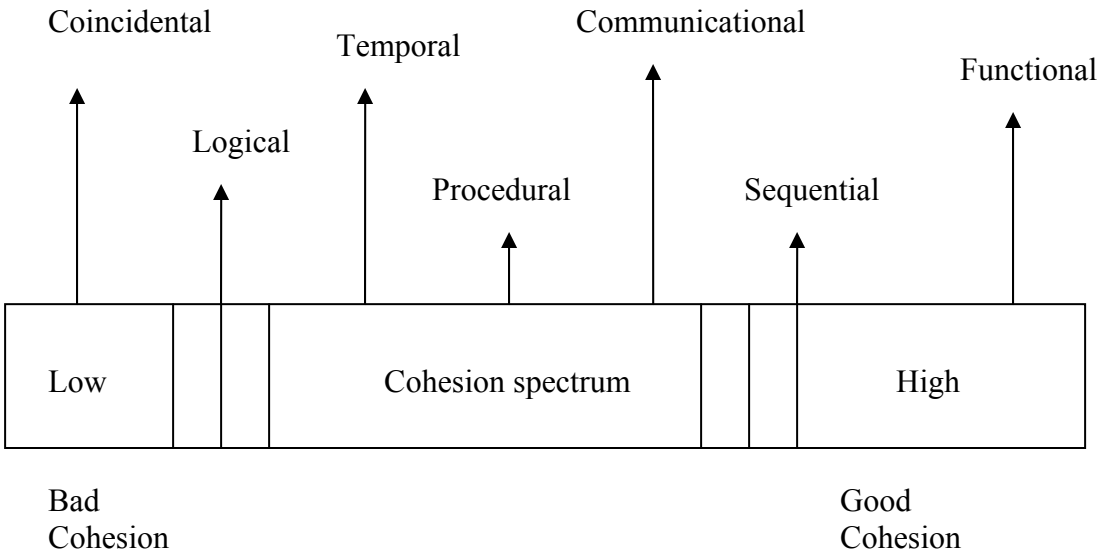

Figure 1.1: Types of Cohesion

1.3 Package Cohesion

A package defines a collection of classes which are conceptually similar or are dedicated to a similar purpose. A cohesive package refers to the self-containedness of a unit, the

degree to which it is a cohesive and the degree to which it can reasonably reduce its dependencies on other units. This is what is referred to as encapsulation [5].

Encapsulation implies both separation and unification. There is a separation of concerns, often expressed as the separation between a meaningful interface and a detailed implementation of some kind. There is also unification: as the term *encapsulation* suggests, features are combined in a *Package*.

In the more general sense of the word, we can think of packaging in terms of grouping, whether in terms of deployable components, whole subsystems, architectural layers, etc. Each packaging unit should be organized around a single and easily identifiable concept.

When designing systems, we strive to achieve a high degree of cohesion, designing elements that focus on performing a single task. This is mainly because of the following two reasons [6]

- **Reuse.** The ability to reuse existing components to create a more complex system.
- **Evolution.** By creating a system that is highly componentized, the system is easier to maintain. In a well-designed system, the changes will be localized, and the changes can be made to the system with little or no effect on the remaining components

Designing cohesive packages means creating packages that offer coarse-grained, yet very focused, behaviors. Also creating highly cohesive packages offers significant advantages. For instance, packages are independently deployable. A package independent of any other package can be deployed as a reusable unit as discussed above. The size and

complexity of systems built using components necessitates some form of modeling. This is needed in order to comprehend the systems, communicate the design to others and to help manage the development process. The Unified Modeling Language (UML) has made explicit provisions for components in its metamodel.

1.4 UML

The Unified Modeling Language{ XE "Unified Modeling Language" } (UML{ XE "Unified Modeling Language (UML)" }) [7] proposed by OMG{ XE "Object Management Group" } (Object Management Group) is now industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. Its success lies in the fact that it enjoys widespread industry support and methodology-independence. Regardless of the methodology used to perform analysis and design, UML can be used to express the results.

The building blocks of UML include the following [8]:

1. Things
2. Relationships
3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

1.4.1 Things

There are four kinds of things in the UML:

- Structural things
- Behavioral things
- Grouping things

- Annotational things

1.4.2 Relationships

There are four kinds of relationships in the UML:

- Dependency
- Association
- Generalization
- Realization

1.4.3 Diagrams

UML diagrams are simply projections into system; they are used to visualize systems from different perspectives[8]. The UML includes nine diagrams, these are[8][9]:

1.4.3.1 Class diagrams

A Class diagram gives an overview of a system by showing its classes and the relationships among them. Class diagrams are static, they display what interacts but not what happens when they do interact. A Class is divided into two main parts, attributes and method along with their parameters. Association{ XE "Association family" } (Generalization{ XE "Generalization" }, Aggregation{ XE "Aggregation" } and Simple Association) between classes is also shown in the class diagram. Object diagrams show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships. Figure shows a simple example of class diagram that we will use throughout this chapter.

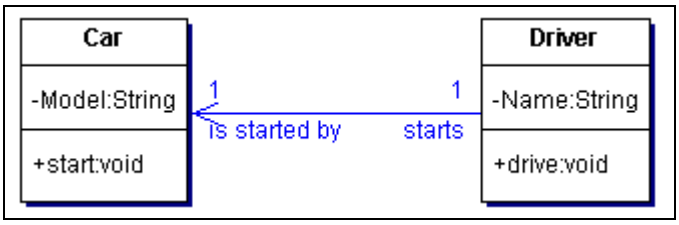

Figure 1.2: UML{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" } Class diagram example

1.4.3.2 Sequence diagrams{ XE "Sequence diagrams" }

Class and object diagram{ XE "object diagram" }s are static model views. Interaction diagrams are dynamic. They describe how objects collaborate. A sequence diagram is an interaction diagram that details how operations are carried out, what messages are sent and when. Sequence diagrams{ XE "Sequence diagrams" } are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence. A Message{ XE "Message" } is a specification of Stimulus; i.e., it specifies the roles that the sender and the receiver Instances should conform to, as well as the Procedure that will, when executed, dispatch a Stimulus that conforms to the Message. The predecessor is a comma-separated list of sequence numbers{ XE "sequence numbers" } followed by a slash (‘/’): sequence-number ‘,’ . . . ‘/’

The clause is omitted if the list is empty. Each sequence-number is a sequence-expression without any recurrence terms. It must match the sequence number of another Message{ XE "Message" }. The meaning is that the Message is not enabled until all of the communications whose sequence numbers{ XE "sequence numbers" } appear in the list has occurred. Therefore, the list of predecessors represents a synchronization of threads. Note that the Message corresponding to the numerically preceding sequence number is an

implicit predecessor and need not be explicitly listed. All of the sequence numbers with the same prefix form a sequence. The numerical predecessor is the one in which the final term is one less. That is, number 3.1.4.5 is the predecessor of 3.1.4.6. Sequence diagrams{ XE "Sequence diagrams" } have two dimensions: 1) the vertical dimension represents time and 2) the horizontal dimension represents different instances or roles. Messages in sequence diagrams are ordered according to a time axis. This time axis is usually not rendered on diagrams but it goes according to the vertical dimension from top to bottom. Sequence diagrams do not use sequence numbers like collaboration diagrams{ XE "collaboration diagrams" } to represent the message ordering. Message ordering is performed by the time axis.

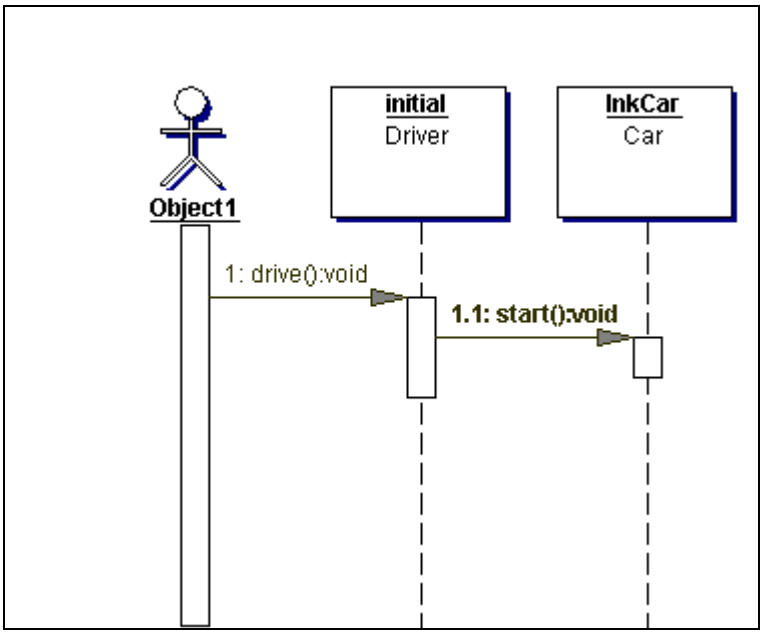

Figure 1.3: UML{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" } Sequence Diagram simple example

1.4.3.3 Collaboration diagrams

Collaboration diagrams are also interaction diagrams. They convey the same information{ XE "processing systems" } as sequence diagrams, but they focus on object roles instead of the times that messages are sent, where as in a sequence diagram object roles are the vertices and messages are the connecting links. One can be generated from other and some case tools like Together { XE "Tools: Together" }[10] provide this functionality. Figure 1.3 shows the previous sequence diagram as collaboration diagram.

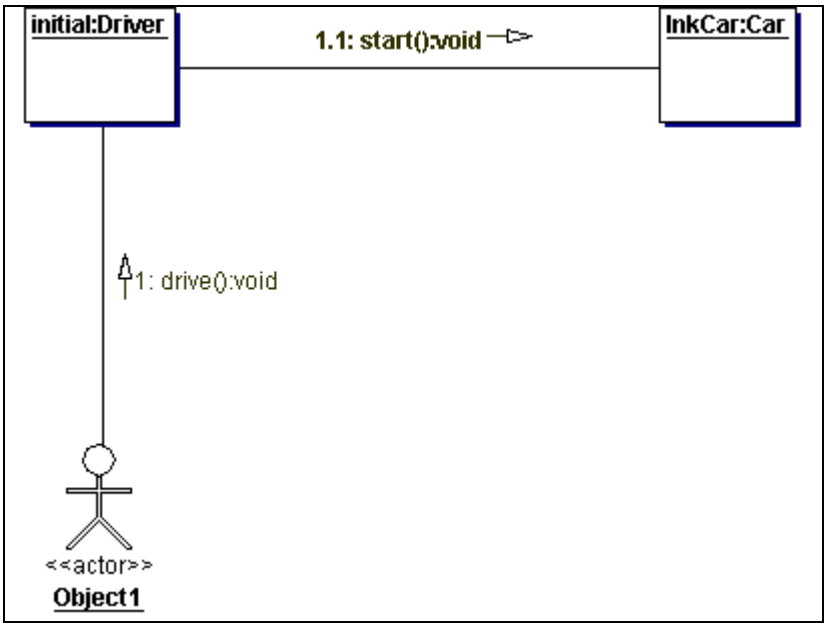

Figure 1.4: UML{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" } Collaboration diagram for Simple example

1.4.3.4 Use case diagrams

Use case diagrams describe what a system does from the standpoint of an external observer. The emphasis is on what a system does rather than how. Use case diagrams are closely connected to scenarios. A scenario is an example of what happens when someone interacts with the system.

1.4.3.5 Statechart diagrams

Objects have behaviors and state. The state of an object depends on its current activity or condition. A statechart diagram shows the possible states of the object and the transitions that cause a change in state.

1.4.3.6 Activity diagrams

An activity diagram is essentially a flowchart with some additional features. Activity diagrams and statechart diagrams are related. While a statechart diagram focuses attention on an object undergoing a process{ XE "process" } (or on a process as an object), an activity diagram focuses on the flow of activities involved in a single process. The activity diagram shows the how those activities depend on one another.

1.4.3.7 Component diagrams

A component is a code module. Component diagrams are physical analogs of class diagram. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations. A component diagram describes the organization of the physical components in a system.

1.4.3.8 Deployment diagrams

Deployment diagrams show the physical configurations of software and hardware. They are related to component diagrams in that a node typically encloses one or more components. A UML deployment diagram depicts a static view of the run-time configuration of processing nodes and the components that run on those nodes. In other words, deployment diagrams show the hardware for your system, the software that is

installed on that hardware, and the middleware used to connect the disparate machines to one another.

1.5 Motivation

Software engineering researchers have attached importance to having high cohesion in the modules of software products. They have asserted that highly cohesive program components are desirable because they lead to better external attributes such as reusability, comprehensibility and maintainability. According to Fenton [1], designs that possess high module cohesion and low module coupling are assumed to lead to more reliable and maintainable code. Object-oriented measurement has become an increasingly popular research area. This is substantiated by the fact that recently proposed in the literature are (i) several different frameworks for coupling and cohesion and (ii) a large number of different measures for object-oriented attributes such as coupling, cohesion, and inheritance [36]. While this is to be welcomed, but the matter of concern is that these metrics for cohesion are at the class level and there has been no work on the measurement of cohesion at the package level. Creating highly cohesive packages offers some significant advantages like independent deployment of packages .A package independent of any other package can be deployed as a reusable unit. Classes, on the other hand, can only be deployed with their containing package. If two classes exhibit a high degree of coupling, these two classes are likely to be frequently used together to provide a cohesive set of services. By considering the coupling between classes when designing your packages, you can also minimize dependencies between packages, making it less likely that changes to one package will impact other packages. Therefore, package cohesion assessment is important in terms of maintainability and reusability. But unfortunately

there is no work in this regard and this is what motivated us to come up with a metric that assesses the cohesiveness of a package.

1.6 Main Contributions

The main contributions of this work are:

- Conducting a literature survey of some of the class cohesion metrics and also a survey of the package cohesion principles and reviewing the only existent package cohesion metric paper.
- Proposing a new package cohesion metric
- Validating the proposed metric theoretically as well as empirically by running experiments on varied systems.

1.7 Organization of the thesis

The rest of this thesis is organized as follows. Chapter 2 presents literature survey of class cohesion metrics and the package cohesion principles and critical review of the package cohesion metric in literature.{ XE "coupling" } Chapter 3 presents the OOMeter tool that has been developed at KFUPM. OOMeter has been used for experimentation purpose in this thesis. Chapter 4 explains the new developed package cohesion metric and also its theoretical validation.{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" }{ XE "coupling metrics" }{ XE "XMI" } Chapter 5 presents the empirical validation of our metric and discusses the results.{ XE "UML models" }{ XE "theoretical validation" } Chapter 6 gives the conclusion and future work.

Chapter 2

Literature Survey

Each packaging unit should be organized around a single and easily identifiable concept. However, the concept should be cohesive rather than coincidental. In this chapter we first begin with some definitions of package or component cohesion and then we present some principles which are basic to package cohesion. We also look at the guidelines which help in designing cohesive packages. Finally, we look at the literature involving package cohesion. It is worth mentioning here that as discussed above very little work has been done in assessing the cohesiveness of a package, therefore we found very little literature on package cohesion. We hope that more people come forward and explore this very important area to which we lay some foundation step.

2.1 Definitions

The basic definition of package cohesion is a collection of classes which are conceptually similar or are dedicated to a similar purpose. Most of the definitions speak more or less about the same thing. Let us take a look at how some of the researchers defined package or component cohesion. Hopkins [6] defined it as "A software component is a physical packaging of executable software with a well-defined and published interface." D'Souza and Wills [11] define component as "A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger"

Similarly, Szyperski provides the following definition: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [12].

2.2 Background

By definition, cohesion captures the degree of interdependence among elements of the same module [13]. As explained in chapter 1, modules with strong cohesion are easier to maintain and they greatly improve the possibility of reuse. The definition of cohesion can have two interpretations; a module is said to be cohesive if (i) its elements are tailored towards one functionality and (ii) the module is self-content as in it does not need to rely on other modules for its function to be achieved. The programming paradigm in question determines what a module is and what an element is. In procedural paradigm, elements of module are statements, sub functions etc. In object-oriented paradigm, the counterparts of module are classes and methods. The elements of a method are statements and attributes since they are accessed either directly or via access functions in the methods. The elements of an object class are methods and instance variables In the following sections we discussed some of the proposed metrics in procedural paradigm and those in object-oriented paradigm found in the literature related to class cohesion. We do not have any metrics for measuring the cohesion of a package. The principles and a naïve paper on package cohesion are discussed in the later sections. The following sections discuss about the cohesion metrics for a single class i.e., *class level cohesion* for procedural as well as object oriented paradigms.

2.2.1 Components and Packages

Often there is a confusion regarding what a component constitutes of and what a package contains, because to some class can be a package or a group of classes can be called a package similarly a component can be a single package or a group of packages tailored to achieve a single functionality as is the case for a package but at a more fine grained level than a component. In this section we try to list out whether the literature provides a clear distinction between a package & a component, so in this sub section we first list out the definitions that we found in literature regarding a package and component and then try to figure out whether we can reach at a consensus regarding the boundary between the two.

According to ngyuen [41], a *component* is basically an entity that represents a *logical object* in a software system. It can be versioned, saved, loaded, and exists within the version space of a software system. A component is designed to represent a *coarse-grained* abstraction and can belong to multiple projects. It is *not* designed to model fine-grained objects or abstractions, though it is not prohibited to do so. A component can be used to model logical abstractions at the design level such as design modules, architectural components, UML diagram entities, data flow diagram entities, ER diagram entities, subsystems, etc. At the implementation level, components can model programs, object-oriented classes, functions, packages, files, directories, documentation, etc." Odell [42] states that "UML describes two ways of expressing aggregation for OO structure and behavior: *components* and *packages*. Components are physical aggregations that compose classes for implementation purposes. Packages aggregate modeling elements into conceptual wholes. Here, classes can be conceptually grouped for any arbitrary purpose,

such as a subsystem grouping of classes." According to Nathan la belle "A package is a bundle of related components necessary to compile or run an application."

D'Souza and Wills define component as "A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger" [11].

The following definitions of "software component" typify those emerging in the software industry.

A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces. (Philippe Krutchen, Rational Software)

A runtime software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime. (Gartner Group)

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [12]

A business component represents the software implementation of an "autonomous" business concept or business process. It consists of the software artifacts necessary to express, implement, and deploy the concept as a reusable element of a larger business system. (Wojtek Kozaczynski, SSA)

In UML however the term Package is used to represent a collection of classes.

From the above definitions it is clear that it is difficult to make a defined boundary between a package and component. As they tend to use these terms in their own contexts relevant to their own perceptions. According to our understanding a component is at a more abstract level than a package and in a sense it embodies package/packages to provide functionality. Packages on the other hand are a collection of classes that work together to realize the component's functionality.

2.3 Cohesion Metrics in procedural programs

A procedural programs is composed of one or more units or modules and each module is composed of one or more procedures (e.g of procedural programs include C, FORTRAN etc).

2.3.1 The SFC and WFC Metrics

Bieman and Ott [14], proposed three cohesion measures, these measures are: Strong Functional Cohesion (SFC(p)), Weak Functional Cohesion (WFC(p)) and Adhesiveness of a procedure (A(p)).

Strong Functional Cohesion (SFC) is defined as the ratio of super-glue tokens to the total number of data tokens in a procedure p. It is given by the following formula.

$$\mathrm{SFC(p)} = \frac{|\ Supergluetokens\ |}{|\ tokens\ |}$$

The SFC is a measure of the minimal functional cohesion in a procedure.

The weak functional cohesion (WFC) is defined as the ratio of glue tokens to the total number of tokens in a procedure p.

$$WFC(p) = \frac{|\ gluetokens\ |}{|\ tokens\ |}$$

The last measure they proposed is *Adhesiveness*, this is related to the number of slices that each token "glues" together. The *Adhesiveness* of a procedure p is defined as follows:

$$A(p) = \frac{\sum_{tokens} \frac{|\ programslicescontainingagluetoken\ |}{|\ programslices\ |}}{|\ tokens\ | * |\ programslices\ |}$$

A *program slice* is a set of program statements which include references to a particular program variable. A *glue token* is a token which is used in more than one program slice that includes a certain statement. A *super glue token* unites all the program slices at some statements. The measures capture the number of program slices having glue or super glue tokens as a proportion of total program slices. Note that a procedure having no cohesion would have no glue tokens. However, a procedure having perfect cohesion would have super glue tokens at every statement.

2.3.2 The DLC and DFC Metric

Bieman and Kang proposed two design level cohesion metrics [15]: DLC (Design Level Cohesion) and DFC (Design Functional Cohesion).

An ordinal scale of cohesion measures is defined: Coincidental, Conditional, Iterative, Communicative, Sequential, and Functional. Each pair of output tokens in a module is evaluated for the strongest cohesion the pair exhibits. The minimum of such value over all output token pairs gives the *Design Level Cohesion* (DLC). *Design Functional*

Cohesion (DFC), on the other hand, is a slice based measure which averages adhesiveness of output token slices corresponding with the interface pints of the module.

2.4 Cohesion Metrics in Object Oriented Programs

In this section we describe the cohesion metrics that were proposed to measure cohesion in object oriented programs.

2.4.1 The Degree of Method and Class Cohesion of Eder et al.

Eder et al. [16] have extended the concept of coupling and cohesion developed oringinally for procedural-oriented systems to object-oriented sytems. They distinguised between three tyes of cohesion in an object-oriented systems: method, class and inheritance cohesion. For each type, various degrees of cohesion are defined. In this section we succinctly explain the degrees of each type of cohesion.

2.4.1.1 Method Cohesion

The elements of a method are statements, local variables and attributes of the method's class. Eder et al. define seven degrees of method cohesion as given below from weakest to strongest [16]: They have been discussed before, refer section 1.2

- Concidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential and,

- Functional

2.4.1.2 Class Cohesion

Class cohesion addresses the relationships between the elements of a clas. The elements of a class are its non-inherited methods and non-inherited attributes. The following are the five degrees of class cohesion from weakest to strongest:

Seperable: the objects of a class represent multiple urelated data abstractions

Multifaceted: the objects of a class represent multiple related data abstractions. The relation is caused by at least one method of the class which useds all these data abstractions.

Non-delegated: there exist attributes which do not describe the whole data abstraction represented by a class, but only a component of it.

Concealed: there exist some useful data abstraction concealed in the data abstraction represented by the class. Consequently, the class includes some attributes and methods which might make another class.

Model: the class represents a single, semantically meaningful concept.

2.4.1.3 Inheritance

This is similar to class cohesion, but it is a bit different in that it considers all methods and attributes in a class including those that are inherited.

2.4.2 The LCOM1 and LCOM2 Metrics

Chidamber and Kemerer [17] use the notion of degree of similarity of methods to propose a cohesion metric, Lack of Cohesion Measure (LCOM). The definition of this metric is given below.

Definition 2.1:

Consider a class *C* with *n* methods $M_1, M_2, \ldots, M_n$. Let $\{I_i\}$ = set of instance variables used by method M_i. There are *n* such sets, i.e., $\{1_i\}, \{I_2\}, \ldots, \{I_n\}$. LCOM1(C) = the number of disjoint sets formed by the intersection of *n* sets.

In other words, LCOM1 is the number of pairs of methods with no common attributes references.LCOM1 is an inverse cohesion measure. A high value of LCOM1 indicates low cohesion and vice versa. In [18], Chidamber and Kemerer have given the following new definition for *LCOM*. Let the new LCOM be LCOM2.

Definition 2.2:

Consider a class *C* with methods $M_1, M_2, \ldots, M_n$. Let $\{I_i\}$ = set of instance variables used by method M_i. There are *n* such sets, i.e., $\{1_i\}, \{I_2\}, \ldots, \{I_n\}$. Let $P = \{ (I_i, I_j) \mid I_i \cap I_j = \varnothing\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varnothing\}$. If all *n* sets $\{1_i\}, \{I_2\}, \ldots, \{I_n\}$ are $\varnothing$ then let $P = \varnothing$.

$$\text{LCOM2} = \begin{cases} |P| - |Q|, if \ |P| > |Q| \\ \\ 0, otherwise \end{cases}$$

In other words, P is the number of pairs of methods without shared instance variables and Q is the pairs of methods with shared instance variables.

2.4.3 The LCOM3, LCOM4 and Co Metrics

Hitz and Montazeri [20] evaluated the metrics suit for object-oriented design put forward by Chidamber and Kemerer in [18] by applying the principle of measurement theory. One

of the metrics evaluated is Lack of Cohesion in Methods (LCOM). They proposed alternative definitions for the LCOM metric[19][20], as presented in the following definitions.

Definition 2.3:

Let X denote a class, I_x the set of its attributes, and M_x the set of its methods. Consider a simple undirected graph $G_x(V, E)$ with $V = M_x$ and $E = \{(m, n) \in V \times V \mid \exists\, I \in I_x\text{: } (m \textit{ accesses } i) \wedge (n \textit{ accesses } i)\}$.

LCOM3(C) = Number of connected components of G_x.

Hitz and Montazeri identified a problem with the *access methods* for LCOM3. An access method provides read or write access to an attribute of the class. Access methods typically reference only one attribute, namely the one they provide access to. If other methods of the class use the access methods, they may no longer need to directly reference any attribute at all. These methods are then isolated vertices in graph G_x. Thus, the presence of access methods artificially decreases the class cohesion as measured by LCOM3. To remedy this problem, Hitz and Montazeri proposed a second version of their LCOM measure. In this version, the definition of G_x is changed as follows: there is also an edge between vertices representing methods m_1 and m_2, if m_1 invokes m_2 or vice versa.

Definition 2.4:

Let X denote a class, I_x the set of its attributes, and M_x the set of its methods. Consider a simple undirected graph $G_x(V, E)$ with $V = M_x$ and $E = \{(m, n) \in V \times V \mid (\exists\, I \in I_x\text{: } (m \textit{ accesses } i) \wedge (n \textit{ accesses } i)) \vee (m \textit{ invokes } n) \vee (n \textit{ invokes } m)\}$.

LCOM4(C) = Number of connected components of G_x.

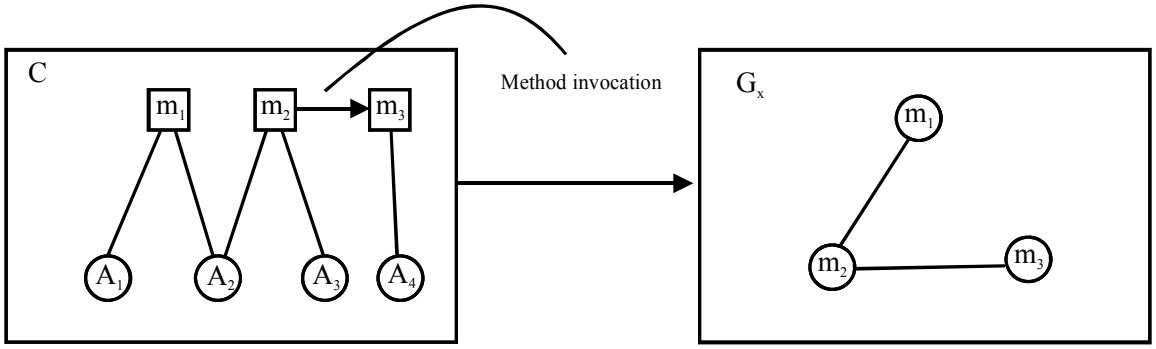

Figure 2.1: A class C and G_x

In the case where G_x consists of only one connected component, i.e., LCOM = 1, the number of edges $|E|$ ranges between $|V| - 1$ (minimum cohesion) and $|V|.(|V| - 1)/2$ (maximum cohesion). Hitz and Montazeri defined a measure *C* ("connectivity") [19] which further discriminates classes having LCOM = 1 by taking into account the number of edges of the connected component.

Hitz and Montazeri defined *C* (Let it be Co in order to differentiate the measure from C used for classes in our examples) as follows:

$$\text{Co(c)} = 2 \cdot \frac{|E_c| - (|V_c| - 1)}{(|V_c| - 1).(|V_c| - 2)}$$

Where E_c and V_c are the edges and vertices of the connection graph of the class *c*.

From the example given in Figure , we have $E_c = 2$ and $V_c = 3$. Hence, Co(C) = 0

2.4.4 The TCC and LCC Metrics

The approach by Bieman and Kang [21] also is based on that of Chidamber and Kemerer's. They also consider pairs of methods that use common attributes. They have defined two different cohesion measures based on the direct and indirect connectivity

between pairs of methods. Two methods that use one or more common attributes are said to be *directly connected*. Whereas two methods that are connected through other directly connected methods are called *indirectly connected*. The indirect connection relation is the transitive closure of the direct connection relation. Thus, a method M_1 is indirectly connected with a method M_n if there is a sequence of methods M_2, M_3, … M_{n-1} such that $M_1 \, \delta \, M_2$, … $M_{n-1} \, \delta \, M_n$.

Where $M_i \, \delta \, M_j$ represents a direct connection.

Let $NDC(C)$ be the number of pairs of directly connected methods of a class C, $NIC(C)$ be the number of pairs of indirectly connected methods of C and $NP(C)$ be the maximum possible number of connections in C. It is clear that for a class with N methods, $NP(C) = N\,(N-1) / 2$.

Tight Class Cohesion (*TCC*) is defined to be a ratio of the number of pairs of directly connected methods in a class, $NDC(C)$, to the maximum possible number of connections in a class, $NP(C)$.

$$TCC(C) = \frac{NDC(C)}{NP(C)}$$

Loose Class Cohesion (*LCC*) is defined to be a ratio of the sum of the number of pairs of directly connected methods, $NDC(C)$, and number of pairs of indirectly connected methods, $NIC(C)$, in a class C to the maximum possible number of connections in C, $NP(C)$.

$$LCC(C) = \frac{NDC(C) + NIC(C)}{NP(C)}$$

With respect to inheritance, Bieman and Kang have stated three options for the analysis of cohesion of a class [21]:

Exclude inherited methods and inherited attributes from the analysis, or

Include inherited methods and inherited attributes in the analysis, or

Exclude inherited methods but include inherited attributes.

Bieman and Kang identified a problem with constructor methods for *TCC* and *LCC*. A class constructor is an initialization function. It generally accesses all attributes in the class, and thus, shares attributes with virtually all other methods. Constructors create connections between methods even if the methods do not have any other relationships. Therefore, the presence of a constructor method artificially increases cohesion as measured by *TCC* and *LCC*. Bieman and Kang have therefore recommended excluding constructors (and also destructors) from the analysis of cohesion [21].

2.4.5 The LCOM5 Metric

Henderson-Sellers et al [22] also based their work on the metric suite of Chidamber and Kemerer [18]. The suite is evaluated from a mathematical point of view and a new formulation for the LCOM measure was defined. Their definition is based on the following properties:

The measure yields 0, if each method of the class references every attribute of the class (this situation is called “perfect cohesion” by Henderson-Sellers”).

The measure yields 1, if each method of the class references only a single attribute.

Values between 0 and 1 are to be interpreted as percentages of the perfect value.

We call their definition LCOM5 and it is defined as follows:

Definition 2.5:

Consider a set of methods $\{M_i\}$ ($i = 1,\ldots..,\ m$) of a class C accessing a set of attributes $\{A_j\}$ ($j = 1,\ldots..,\ a$). Let the number of methods which access an attribute A_j be $\mu(A_j)$ and total number of attributes in $\{A_j\}$ is a.

$$\text{LCOM5} = \frac{\frac{1}{a}\sum_{j=1}^{a}\mu(A_j) - m}{1 - m}$$

2.4.6 The RCI Metric

Briand et al. proposed a cohesion measure in [23] that is based on the visualization of a class as a collection of *data declarations* and *methods*. Data declarations are (i) local type declarations, (ii) the class itself (as an implicit public type), and (iii) public/private attributes (including constants). Briand et al. defined two types of interactions, *DD-interactions* (declaration-declaration interactions) and *DM-Interactions* (declaration-method interactions).

DD-interaction: A data declaration *a* DD-interacts with another data declaration *b*, if a change in *a*'s declaration or use may cause the need for a change in *b*'s declaration or use. We say that there is a *DD-interaction* between *a* and *b*. The following are examples of *DD-interactions*:

If the definition of a type *t* uses another public type *t'*, there is a DD-interaction between *t'* and *t*.

If the definition of a public attribute *a* uses a public type *t*, there is a DD-interaction between *t* and *a*.

If a public attribute *a* is an array and its definition uses public constant *a'*, there is a DD-interaction between *a'* and *a*.

DD-interactions need not be confined to one class. There can be DD-interactions between attributes and types of different classes. The DD-interaction relationship is transitive. If *a* DD-interacts with *b* and *b* DD-interacts with *c*, then *a* DD-interacts with *c*.

DM-interaction: Data declarations can also interact with methods. There is a *DM-interaction* between a data declaration *a* and method *m* either

if *a* DD-interacts with at least one data declaration of *m* (Data declarations of methods include their parameters, return type and local variables), or

if *a* is an attribute and *m* uses/accesses it.

Briand et al. defined *CI*(*C*) (*CI* for cohesive interactions) to be the set of all DD- and DM-interactions present in the class *C* and *Max*(*C*) to be the set of all possible DD- and DM-interactions that can be established in class *C*. *RCI* can be defined as follows:

$$RCI(C) = \frac{|\,Cl(C)\,|}{|\,Max(C)\,|}$$

2.4.7 The CAMC Metric

In 1999, Bansiya et al. [24]proposed a design metric to evaluate cohesion among methods of a class early in the analysis and the design phase. The metric evaluates the consistency of methods in a class' interface using the parameter lists of the methods. The metric can be applied on class declarations that only contain method prototypes (method types and parameter types). They call their metric CAMC (Cohesion Among Methods of Classes).

The CAMC metric measures the extent of intersections of individual method parameter types lists with the parameter type list of all methods in the class. To compute the CAMC metric value, an overall union (T) of all object types in the parameters of the methods of a class is determined. A set M_i of parameter object types for each method is

also determined. An intersection (set P_i) of M_i with the union set T is computed for all methods in the class. A ratio of the size of the intersection (P_i) set to the size of the union set (T) is computed for all methods. The summation of all intersection sets P_i is divided by product of the number of methods and the size of the union set T, to give a value for the CAMC metric. Mathematically, the metric is defined as follows:

$$CAMC = \frac{\sum_{i=1}^{n} | P_i |}{| T | \times n}$$

Where

N is number of methods in the class

M_i is the set of parameters of method *i*

T is the union of M_i, for every *i* = 1 to n

P_i is the intersection of set M_i with T i.e. $P_i = Mi \cap T$

The metric value ranges between 0 and 1.0. A value of 1.0 represents maximum cohesion and 0 represents a completely un-cohesive class.

2.4.8 The CBMC Metrc

In 2000, Chae highlighted two problems with the existing cohesion metrics.They noted that the existing cohesion measures DO NOT [25]:

Take into account the properties of special methods like access methods, constructors etc. thus fail to properly reflect the actual cohesiveness of classes.

Consider the patterns of the interactions among members, they are simply based on counting the number of the instance variables refernced by methods or the number of method pairs with shared instance variables.

In order to cope with these problems, they proposed a new metric called CBMC (Cohesion Based on Member Connectivity) whose definition is given below. Their metric is based on two things: *connectivity factor* and *structure factor*.

Definition 2.6:

The CBMC for a class C, CBMC(C), is defined to be the connectibity factor of its reference graph, $F_c(G_r(C))$, *scaled by the structure factor of its reference graph*, $F_c(G_r(C))$

$$CBMC(C) = F_c(G_r(C)) \times F_s(G_r(C)) = F_c(G_r(C)) \times \frac{1}{n}\sum_{i=1}^{n} CBMC(G_r^i(C))$$

Where $F_c(G_r(C)) = \frac{|M_g(G_r)|}{|M_n(G_r)|}$ is the connectivity factor(represents the degree of the connectivity among the members).

And $F_s(G_r(C)) = \frac{1}{n}\sum_{i=1}^{n} CBMC(G_r^i)$ is the structure factor

M_g and M_n are the set of glue methods and normal methods respectively. Glue methods are the minimum number of methods without which the reference graph will be divided into sub-graphs. G_r^i is one of the *n* children of G_r in the *structure tree*; CBMC denotes the cohesion of a component G_r^i.

$$CBMC(C) = F_c(G_r(C)) \times F_s(G_r(C)) = F_c(G_r(C)) \times \frac{1}{n}\sum_{i=1}^{n} CBMC(G_r^i(C))$$

2.4.9 The CCM and ECCM Metrics

Jaralla et al. in [26] proposed two cohesion metrics for assessing the extent to which an inheritance hierarchy follows four design principle they discussed in their paper. The metrics are: CCM (Class Connection Metric) and ECCM (Enhanced Class Connection Metric), the difinition of these metrics are give below.

$$CCM(C) = \frac{NC(C)}{NMP(C) \cdot NCC(C)}$$

Where NC(C) is the number of actual connection among the methods of the class, NMP(C) is the number of the maximum possible connections among the methods of the class C and NCC(C) is the number of connected components of the connection graph G_c.

$$ECCM(C) = \frac{NC(C)}{NMP(C).NCC(C)} \times (1 - PenaltyFactor(C))$$

or simply,

$$ECCM(C) = CCM(C) \times (1 - PenaltyFactor(C))$$

Where $PenaltyFactor(C) = \frac{NORM(C)}{NOIM(C)}$

NORM(C) is the number of re-implemented methods and NOIM(C) is the number of inherited methods.

2.4.10 The OCC and PCC

Aman et al. [27] proposed two cohesion metrics that not only consider the connections among the component of a class but also consider the sizes of connected modules as well as the strength of method connection,. These metrics are: OCC (Optimistic Class Cohesion) and PCC (Pessimistic Class Cohesion).

Definition 2.7: Weak-connection graph

Given a class, let M be the set of methods, and A be the set of attributes, within the class. A weak-connection graph is defined as an undirected graph $G_w(V, E)$, where

V = M and

E = {{*u*, *v*} Є M x M | ∃ a Є A s.t. (ac (*u*, a) Λ ac (*v*, a))}……(1)

Definition 2.8: Strong-connection graph

Given a class, let M be the set of methods, and A be the set of attributes, within the class. Strong-connection graph is defined as a directed graph G_s (V, E), where V = M and

E = {{*u*, *v*} Є M x M | ∃ a Є A s.t. (*wr* (*u*, a) Λ *re* (*v*, a))}.....(2)

Definition 2.9: Optimistic Class Cohesion (OCC)

Given a class, let M be the set of methods, and A be the set of attributes, within C. Consider the weak-connection graph G_w(V, E), where V = M and E is as given in equation 1. Let n = |M|. For each method m_i Є M (i = 1, . . . , n), let $R_w(m_i)$ be the set of methods which are reachable by m_i on G_w(V, E):

$$\mathrm{Rw(mi)} = \{mj \in M \mid \exists mk1,...,mkp \in Ms.t\{mks, mks+1\} \in E(s = 1,..., p-1), mi = mk1, mj = mkp, i \neq j\}$$

The Optimistic Class Cohesion (OCC) for a class C is defined as follows:

$$OCC(C) = \begin{cases} \max\limits_{i=1,...,n}\left[\dfrac{|\,Rw(mi)\,|}{n-1}\right], (n>1) \\ 0, (n=1) \end{cases}$$

Definition 2.10: Pessimistic Class Cohesion (PCC)

Given a class C, let M be the set of methods, and A be the set of attributes, within C. Consider the strong-connection graph G_s(V, E), where V = M and E is as in equation 2. Let n = |M|. For each method m_i Є M (i = 1, . . . , n), let $R_s(m_i)$ be the set of methods which are reachable by m_i on G_s(V, E):

$$\mathrm{Rs(mi)} = \{mj \in M \mid \exists mk1,...,mkp \in Ms.t\{mks, mks+1\} \in E(s = 1,..., p-1), mi = mk1, mj = mkp, i \neq j\}$$

The Pessimistic Class Cohesion (PCC) for a class C is defined as follows:

$$PCC(C) = \begin{cases} \max\limits_{i=1,...,n}\left[\dfrac{|\,Rs(mi)\,|}{n-1}\right], (n>1) \\ 0, (n=1) \end{cases}$$

2.5 Package Cohesion Principles

There are five principles of package cohesion, namely, Release Reuse Equivalency principle, Common closure principle, the common reuse principle, Acyclic Dependencies Principle and the Stable dependencies principle. We here concentrate on the first three principles as they have been widely considered as the main three principles of package cohesion. These three principles are mutually exclusive. They cannot simultaneously be satisfied. That is because each principle benefits a different group of people. The REP and CRP facilitate reuse, whereas the CCP facilitates maintenance. The CCP strives to make packages as large as possible (after all, if all the classes live in just one package, then only one package will ever change). The CRP, however, tries to make packages very small.

Fortunately, packages are not fixed in stone. Indeed, it is the nature of packages to shift and jitter during the course of development. Early in a project, architects may set up the package structure such that CCP dominates and development and maintenance is aided. Later, as the architecture stabilizes, the architects may refactor the package structure to maximize REP and CRP for the external users to facilitate reuse.

2.5.1 Release Reuse Equivalency Principle:

THE GRANULE OF REUSE IS THE GRANULE OF RELEASE [28]:

A reusable element, be it a component, a class, or a cluster of classes must be managed by a release system of some kind. Users will be unwilling to use the element if they are forced to upgrade every time the author changes it. Clients will refuse to reuse an element unless the author continues to keep track of version numbers, and maintain old versions for a while.

Therefore, one reason for grouping classes into packages is reuse. Since packages are the unit of release, they are also the unit of reuse. Therefore architects would do well to group reusable classes together into packages. The users cannot reuse anything that is not also released. When users reuse something in a released library, he is a client of the entire library.

The REP states that the granule of reuse can be no smaller than the granule of release. Anything that is reuse must also be released. Clearly, packages are a candidate for a releasable entity. It might be possible to release and track classes, but there are so many classes in a typical application that this would almost certainly overwhelm the release tracking system. Some larger scale entity to act as the granule of release; and the package seems to fit this need rather well.

2.5.2 Common Closure Principle

CLASSES THAT CHANGE TOGETHER, BELONG TOGETHER [28]:

A large development project is subdivided into a large network of interrelated packages. The work to manage, test, and release those packages is non-trivial. The more packages that change in any given release, the greater the work to rebuild, test, and deploy the release. Therefore we would like to minimize the number of packages that are changed in any given release cycle of the product.

To achieve this, we group together classes that we think will change together. When we group classes that change together into the same packages, then the package impact from release to release will be minimized. The CCP is an attempt to gather in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, such that they almost always change

together; then they belong in the same package. This minimizes the workload related to releasing, revalidating, and redistributing the software.

2.5.3 Common Reuse Principle

CLASSES THAT ARE NOT REUSED TOGETHER SHOULD NOT BE GROUPED TOGETHER [28]:

Classes are seldom reused in isolation. Generally reusable classes collaborate with other classes that are part of the reusable abstraction. This principle helps us to decide which classes should be placed into a package. It states that classes that are not to be reused together should not belong in the same package. If classes that are not used together are grouped together, changes to a class that is not used about will still force a new release of the package, and to go through the effort of upgrading and revalidating. Similarly classes that are reused together should belong to the same package.

Designers strive to achieve a high degree of cohesion in systems. Systems exhibiting high degrees of cohesion consist of elements that are focused on performing individual tasks, where each task contributes to the overall purpose of the system. Systems with lower degrees of cohesion have elements that perform a mishmash of functionality, where the purpose of each individual element is not well defined.

2.5.4 Acyclic Dependencies Principle (ADP)

THE DEPENDENCIES BETWEEN PACKAGES MUST FORM NO CYCLES [28]

Cycles among dependencies of the packages composing an application should almost always be avoided. In other words, packages should form a directed acyclic graph (DAG).

2.5.5 Stable Dependencies Principle (SDP)

DEPEND IN THE DIRECTION OF STABILITY [28]

In the context of software development, stability often is used to describe a system that is robust, bug free, and rich in structure. In a more general sense, stability implies that an item is fixed, permanent, and unvarying.

2.6 Guidelines for designing cohesive packages [29]

Reusability is critical in creating cohesive packages. Here are some of the guidelines for creating cohesive and reusable packages based on the java.io package.

2.6.1 Create cohesive packages

While reuse is typically considered at a class level; it's a fairly uncommon practice to reuse classes individually. Instead, reuse typically involves combining multiple classes to perform a higher-level system function. The java.io package is a wonderful example of a cohesive package. Few classes in this package are used independently. Instead, it's common to use multiple classes together to realize the desired functionality. Cohesive packages ease maintenance and promote reusability because they provide one module form in which fine-grained classes can be assembled to provide coarse-grained functionality. Packages lacking cohesion contain classes that independently perform a myriad of disjointed functions. This makes reuse difficult because you must import classes from multiple packages, and it makes maintenance difficult because you must modify classes in multiple packages. The result is a more painful deployment or redeployment of application services because multiple packages must be compiled, tested and distributed.

2.6.2 Consider the impact of package contents on reuse

When designing packages, always consider other classes that the classes within a package will use. In addition, you should consider classes not necessarily referenced by the contents of a package but that are frequently reused in conjunction with those classes. When creating cohesive packages, you can view packages as coarse-grained reusable components.

2.6.3 Emphasize reusability at the package level

Packages should be collections of classes that are highly cohesive and in which each class likely plays a part in performing a coarse-grained unit of functionality. Such packages are reusable components that abstract away much of the implementation-specific details, making the functional unit easier to use. For this to work effectively, developers who want to reuse a package must clearly understand the package's functionality and how to communicate with the package.

2.6.4 Design packages at a single level of granularity

A package's functionality should be more coarsely grained than the functionality provided by the individual classes in the package. Offering fine-grained services from a coarse-grained module will introduce maintenance challenges. In addition, clients of the package must have a greater understanding of the internal functionality of the package, raising concerns about encapsulation.

2.6.5 Make a package's published interface well known

A package's published interface consists of more than the public methods on the public classes in the package. Any other class in the system importing the containing package can call the contained class's public methods. Any change to a public method involves

updating that method's callers. If the package is used within a single application, it might be easy to determine who the callers are. This problem is compounded, however, if the package is used across applications or has been published to the Web as a service to third parties whom you may not know about. A published interface is a public method that is meant to be called by third-party software: Take great care when defining the published interface. Because a published interface is not distinguished from a public interface, developers should be actively discouraged from calling public methods. Public methods that are routinely called should be published. The only way to ensure future compatibility is to use published interfaces that represent a package's specification, not the implementation.

2.6.6 Maintainability

While reusability is a noble goal of object-oriented application design, you also should focus on the importance of maintainability. By packaging classes appropriately, you can limit changes to system behaviors to fewer packages, resulting in more timely and reliable software modifications.

2.6.7 Tightly coupled classes belong in the same package

Although this guideline emphasizes coupling between classes, placing tightly coupled classes in the same package results in a more cohesive package. If two classes exhibit a high degree of coupling, these two classes are likely to be frequently used together to provide a cohesive set of services. By considering the coupling between classes when designing your packages, you can also minimize dependencies between packages, making it less likely that changes to one package will impact other packages.

2.6.8 Classes that change together belong in the same package

Obviously, classes that are tightly coupled to each other are likely affected similarly by change. Any change to the interface of a class often results in some corresponding changes to all classes that depend upon the modified class; at the least, classes calling the modified method must be changed. You can easily mitigate this change-management risk by placing tightly coupled classes in the same packages.

However, some classes that are not tightly coupled are still jointly affected by a required change to system behavior. In situations such as these, you should place these classes in the same package. Because separate classes need the required change, they may work to provide a coarse grained service, even though they may not be directly coupled. So, if a required change cuts across system classes, these classes should be as closely located to each other as possible.

2.6.9 Classes not reused together belong in separate packages

The strength of the packaging guideline lies in its prohibition on packaging classes that don't offer true cohesion. Even though classes may frequently be reused together, they may not always change together, so you should consider packaging these classes separately. Of course, this may mean importing multiple packages to use the individual classes, which at first seems inflexible. However, upon closer inspection, this approach's advantages become clear. If you emphasize reusability at the package level, creating a dependency to reuse any class in a package results in a dependency upon all classes in the package, albeit indirectly. If a single class in a package changes, that package must be redeployed before the system realizes the benefits of the change. Any change to all other classes in that same package must also be deployed, since deployment minimally occurs

at the package level in Java. The result may be upgrades to individual classes that you aren't interested in upgrading.

Depending on any class in a package creates an indirect dependency on all other classes in the package. When a class in the package changes, even if it isn't being used, the entire package must be released.

2.6.10 Classes not deployed together belong in separate packages

Unfortunately, the object-oriented paradigm does not lend itself to reusing individual classes, despite appearances during the initial development effort. The approach breaks down quickly as the system grows, and maintainability becomes more important. As component versioning and supporting becomes important, packaging classes in separate packages is an effective management tactic.

2.7 Review of Package cohesion metric research paper

"Defining metrics for software components". Giancarlo et.al [30]

This is the only paper that we found in the literature which is close to the problem addressed in this thesis. It focuses on the component oriented programming (which here is the group of classes) and presents a set of metrics by extending the Chidamber and Kemerer metrics suite [18]. But it is just a naïve approach and does not go in depth about the computation of metrics as the authors too admit.

The set of metrics that the authors address are Weighted methods per class (WMC) which is the summation of the complexity of each method of the class, Depth of inheritance (DIT) is the maximum length of the path from the class to the root of the hierarchical tree, Number of children (NOC) is the number of immediate subclasses of a class, Coupling between object classes (CBO) is the count of the classes to which it is

coupled. Here two classes are coupled if one acts on attributes of the other, response for a class (RFC) is the cardinality of the set of methods that can potentially be executed in response to a message received by an object of that class and Lack of cohesion in methods (LCOM) is a measure of how poorly the methods and variables are related in a class. The following are the extensions proposed to the above discussed set of metrics.

Extensions for NOM: weighted classes per component and number of classes.

$$\text{WCC (weighted class per component)} = \sum_{i=1}^{m} NOM(Ci)$$

Where NOM is used to measure the complexity of the classes. If it is considered that the complexity of each single class as unity, we can define the complexity of the component *K* as the *number of classes* (NC). C_i is the no. of classes i.e., a component is formed by m classes C_i , $i \in [1..m]$

Extensions for DIT: maximum of the DIT

Now for the DIT they consider both the highest value of DIT, MAXDIT. The definition is:

$$\text{MAXDIT} = \max_{c_i \in k} \{\text{DIT}(C_i)\}$$ Where K is the component itself.

Extensions for NOC: number of children for a component

NOC is extended by counting the number of children of all the classes in the component; here this metric is called number *of children for a component* (NOCC)

$$NOCC = \sum_{i=1}^{m} NC(Ci)$$

Extensions for CBO: external CBO

It is defined as the level of coupling for a component: *external CBO* (EXTCBO) is the number of external classes coupled to it

$$EXTCBO = \sum_{i=1}^{m} Ei$$

Where Ei is the number of external classes coupled to the class *Ci*.

Extensions for RFC: response set for a component

The *response set of a component* (RFCOM) is the number of all the methods in the member classes and the methods called by those classes. This value is the sum of the values of RFC for all the classes in the set

$$RFCOM = \sum_{i=1}^{m} RFC(Ci)$$

Measuring component cohesion

Here they propose to measure *component cohesion*, as the number of internal classes to which a class is coupled normalized with the number of the possible coupling relationship among the classes: *m(m-1)*.

$$CC = \sum_{i=1}^{m} \sum_{j=1, j\#i}^{m} h(Ci, Cj) / m(m-1)$$

Where $h(C_i , C_j) = 1$ if C_i and C_j are coupled

0 otherwise

But they have not elaborated here as to whether they consider only the method interactions in the meaning of the consideration, C_i and C_j are coupled i.e., two classes have a link Also they normalize it by the possible coupling relationship, this too again is not elaborated as to what this coupling relationship is.

As far as *theoretical validation* of the metrics is concerned the authors have extended the validation criteria's of Briand et.al [31] to support the component measures.

We will discuss the criterion which is of concern to us i.e., Component cohesion (CC) metric validation criteria. The criterions they looked are the properties of nonnegativity and normalization, null value, monotonicity and cohesive modules.

Non-negativity and normalization: CC cannot be negative because it is the sum of the number of internal classes to which a class is coupled. This value has a defined maximum equal to 1, reached when all the classes are coupled among them-selves.

Null value: CC is zero if the classes inside the component are not coupled.

Monotonicity: If we add internal relations inside a component, obviously the CC does not decrease, but it either increases or remains the same.

Cohesive modules: If two modules that do not have relations between them are merged, CC does not increase. In fact, the number of relations among the classes inside the component will be the same.

The authors have tested the proposed set of metrics to the Gamma's design pattern [32] .No significant linear relation has been identified between the proposed metrics and the number of uses of the design patterns.

Chapter 3

Implementation

In this chapter, we discuss a tool that we used for measuring package cohesion. As part of their research work, Master students from KFUPM built a software metrics tool (OOMeter) that captures coupling metrics from UML models stored in XMI [33]. We extended this tool to support package cohesion metrics. In this chapter we give a description of the tool, OOMeter

3.1 Architecture of OOmeter

Architecture of OOMeter{ XE "OOMeter" } can be classified as a heterogeneous architecture in Garlan & Shaw's [34] terminology. Main components are two parsers for java and XMI{ XE "XMI" }, two data repositories for storing source data and metrics output and a front end.

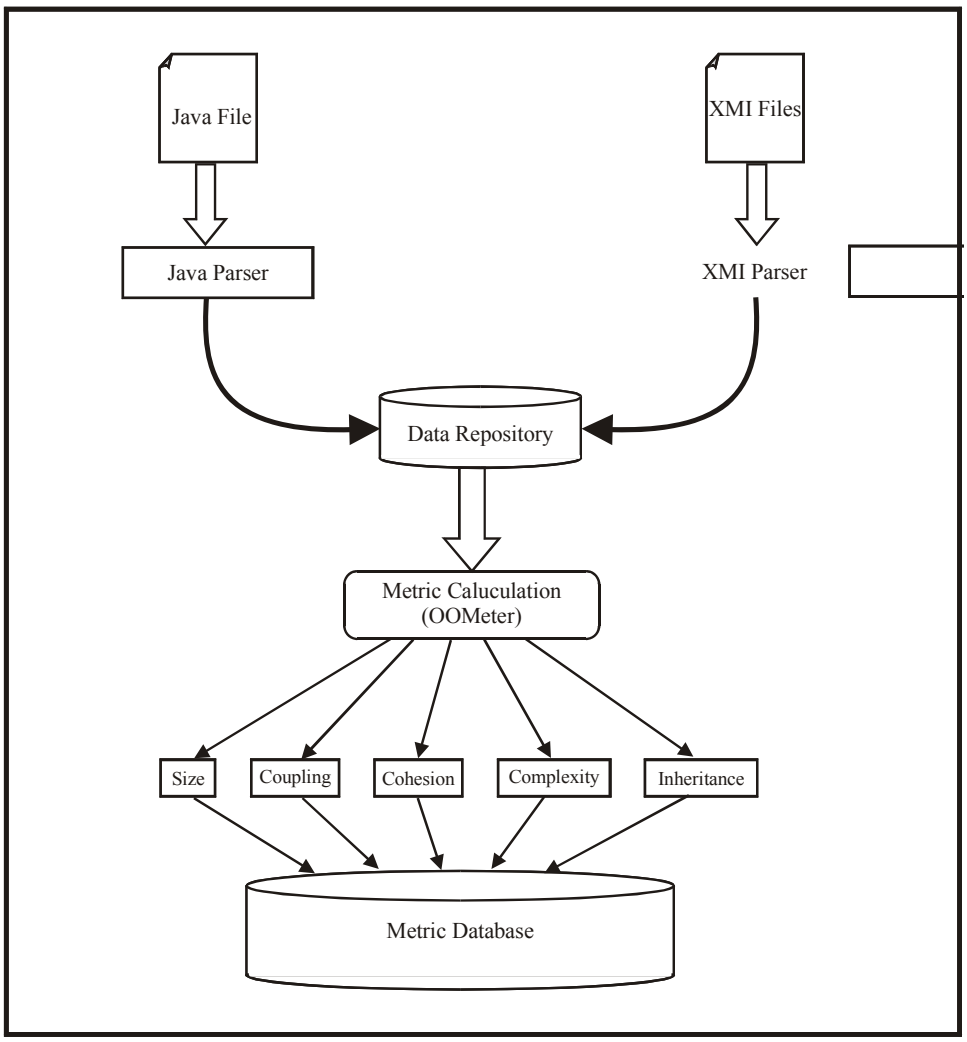

Figure 3.1: Architecture OOMeter{ XE "OOMeter" }

As shown in Figure 3.1, object-oriented systems are parsed to the tool in order to collect the data that can be used in computing the various software metrics supported by the tool. The collected data is stored into a Central Data Repository and the results of the computation of all the metrics are stored in a different database called Metric Database. At the moment the tool supports the parsing of both java source files as well as UML models stored in XMI format. The tool supports the computation of a variety of software measures, which includes size, coupling, cohesion and complexity measures.

The results of XMI{ XE "XMI" } parsing are stored in the data repository in language independent format. It can be observed in the data model shown in Figure 3.2 that it captures all basic information{ XE "processing systems" } that is needed for most of the software metrics.

The data model serves as our language independent repository for storing all the information{ XE "processing systems" } parsed from the projects, whether it may be java source files or UML{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" } models{ XE "UML models" } stored in XMI{ XE "XMI" }.

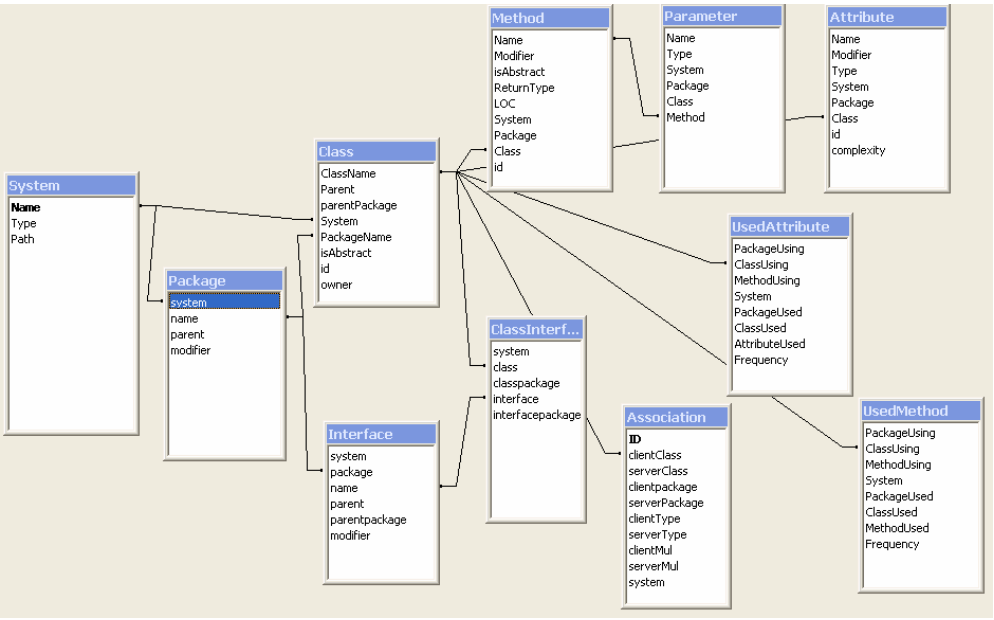

Figure3.2: Data Model of OOMeter{ XE "OOMeter" }

Figure shows the component or package diagram of OOMeter{ XE "OOMeter" }. *OOMeter* is the main package of the system that uses other packages. *XMIParser* and *JavaParser* are the parsers for XMI{ XE "XMI" } and java files respectively. *Common* contains the database handling classes and *XMIParserTest* and *JavaParserTest* are the Junit test classes for the two parsers.

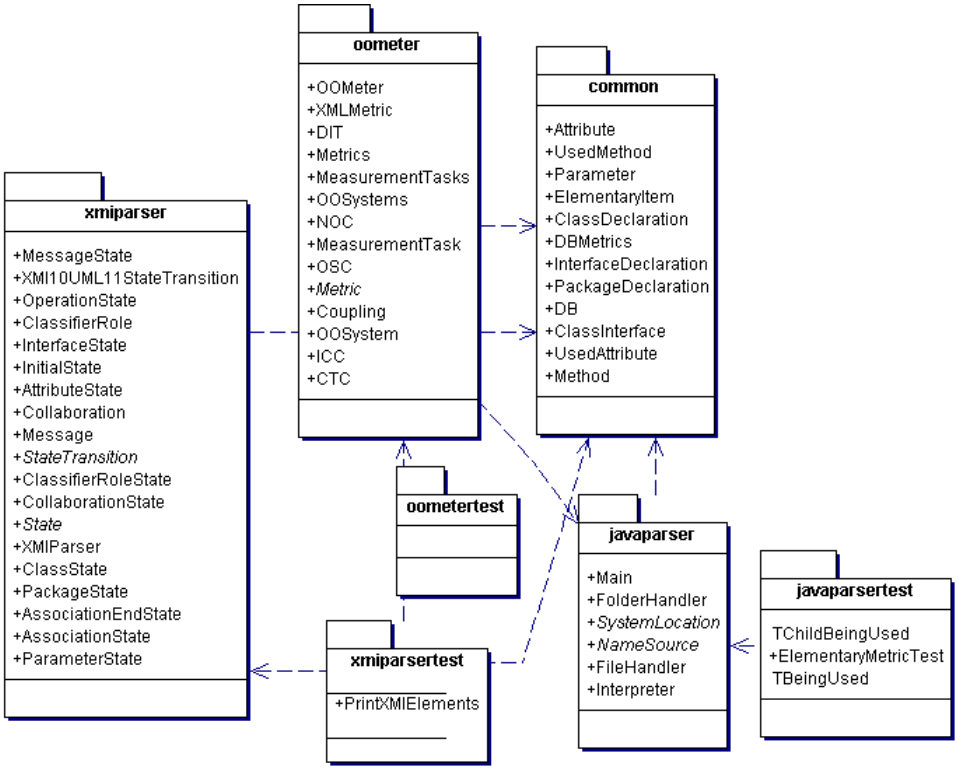

Figure 3.3: Component Diagram of OOMeter

{ XE "OOMeter" }

3.2 Design of XMI{ XE "XMI" } Parser { XE "XMI Parser" }

In this section we will focus on the design of the XMI{ XE "XMI" } parser component of the system, As discussed in the XMI processing{ XE "XMI processing" } approach selection section, we selected a state machine-using SAX{ XE "SAX" }. We used a hybrid of table driven state pattern and sub classing state pattern [35].

The XMI{ XE "XMI" } parser goes through several states that denote the basic entities in the UML{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" } Meta model, capturing in each state all the information{ XE "processing systems" } needed from the child nodes and entity values.

The following state charts show the transitions for each of the UML{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" } diagram types that we take as input.

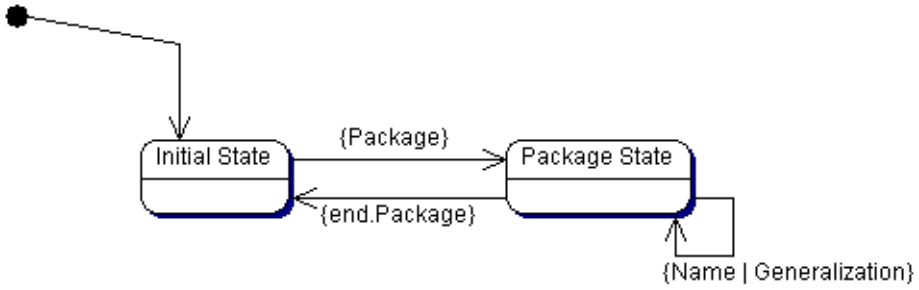

Figure 3.4: State Chart for Package diagram

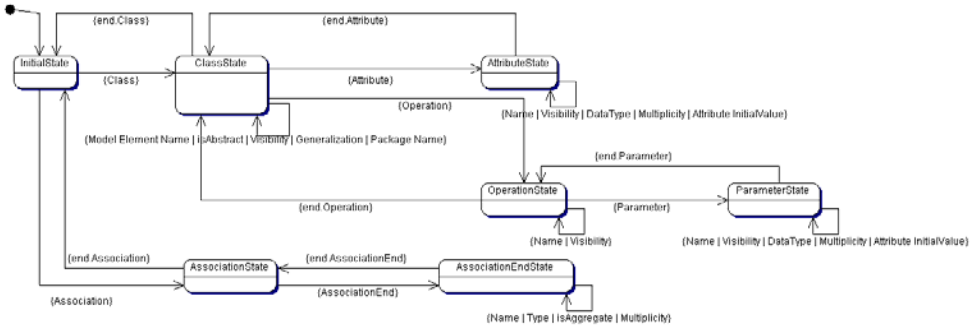

Figure 3.5: State Chart for Class diagram

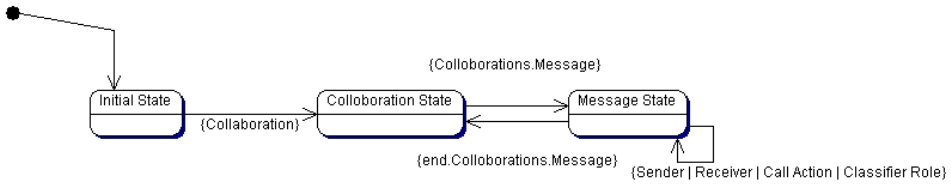

Figure 3.6: State Chart for Sequence diagram

The class diagram below shows all classes of XMI{ XE "XMI" } Parser{ XE "XMI Parser" }.

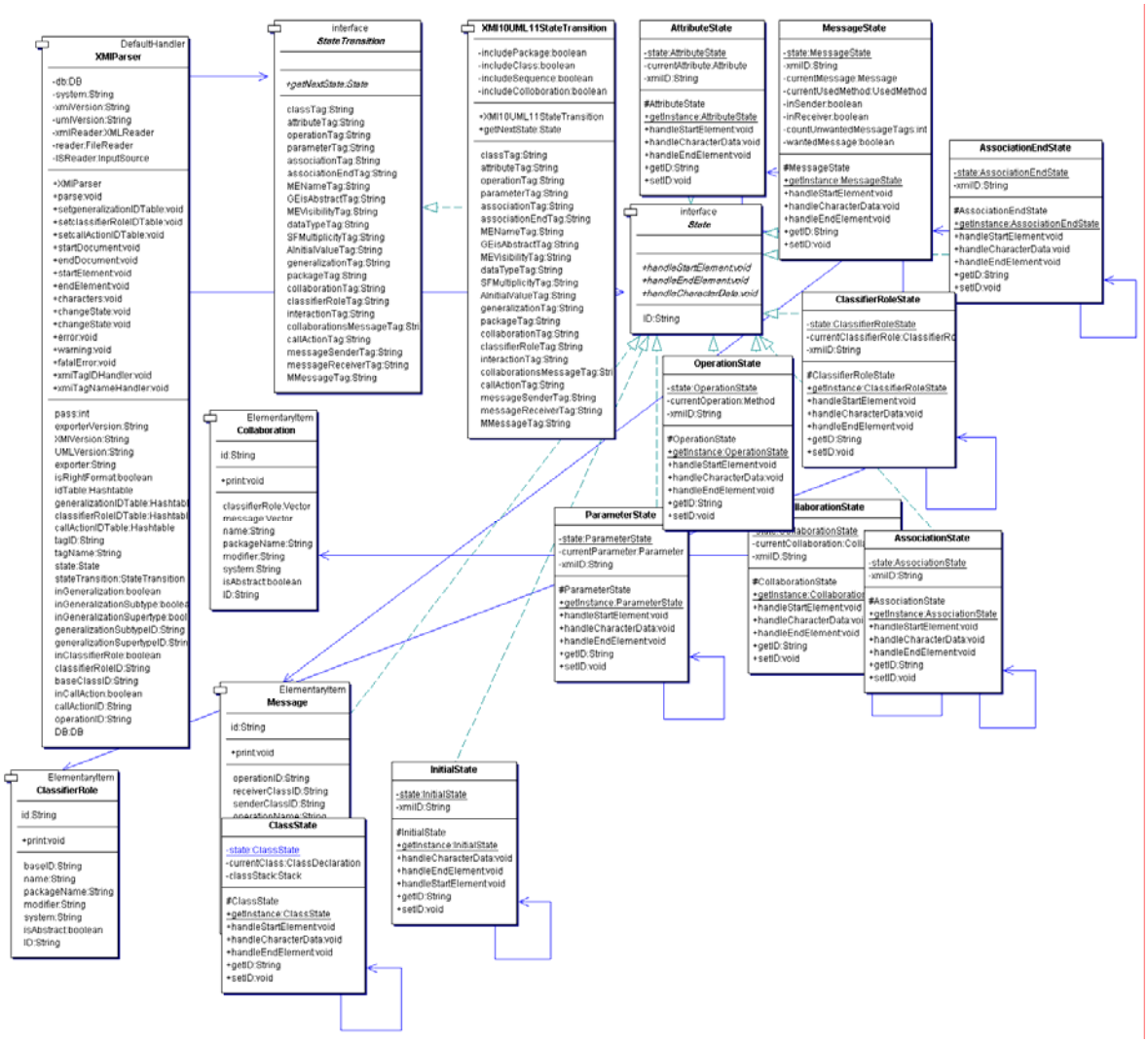

Figure 3.7: Class diagram XMI{ XE "XMI" } Parser{ XE "XMI Parser" }

As show in the class diagram each of the concrete state implements the *state* interface. All the logic of state transitions is encapsulated in XMI{ XE "XMI" } and UML{ XE "Unified Modeling Language (UML)" }{ XE "Unified Modeling Language" } version specific State Transition table *XMI11UML10StateTransition* in this case. This transition table implements the *StataTransition* interface; hence if we want to support another version of XMI or UML other concrete classes can be derived for those versions. The amount of change needed in concrete classes depend on the nature of change in version. For example node names can be easily mapped from XMI 1.0 to XMI 1.1, hence change

is a question of mapping the tag names, but for major changes as in XMI 2.0, this may not be very straightforward.

Chapter 4

Package Cohesion Metric

4.1 New Package Cohesion Metric

In this chapter a new package cohesion metric which takes into account an exhaustively researched set of connections is presented. A metric as a measure should have a simple procedure or process for capturing the software attributes it measures. The result of a metric should be normalized for easy understanding and easy comparison. A metric should also provide an interpretation for the measure (the numerical value). The table below lists the connection types that we are going to use in our metric calculation.

Type of interaction determines the mechanism by which two classes are connected. Different types of interaction have been investigated and the most appropriate ones which would have an effect with respect to package cohesion have been listed.

S.NO	ELEMENT 1	ELEMENT 2	DESCRIPTION
1	Method *m* of class c	Attribute *a* of class c'	*m* references *a*
2	Method *m* of class c	Method *m'* of class c'	*m* invokes *m'* directly
3	Method *m* of class c	Method *m'* of class c'	*m* invokes *m'* indirectly via other methods that directly invoke each other.
4	Method *m* of class c	Method *m'* of class c'	*m* and *m'* directly reference an attribute a of class c'' in common
5	Method m of class c	Method m' of class c'	*m* and *m'* invoke a method m'' of class c'' in common
6	Method m of parent Class c	Method m' of class c'	m' invokes method m of class c
7	Attribute a of parent class c	Method m of Class c'	m references attribute a of class c
8	Method m in class c	Classes c' & c''	Child classes c' &c'' share common method(s) from their parent class c
9	Attribute a of class c	Classes c' & c''	Child classes c' &c'' share common attribute(s) from their parent class
10	Attribute a of class c	Class c'	C' is the type of an attribute of c
11	Method m of c	Class c'	C' is the type of an input or output parameter of a method m of c
12	Method m of c	Class c'	C' is the type of a local variable of a method m of c

Table 4.1: Connection Types

The connections have been categorized into three groups. The first one captures the interaction between the methods and attributes of two classes in a package. Here we also consider indirect connections between methods as listed in connection type 3. The connection types 1 to 5 correspond to this group. The second category, type 6 to 9, captures the connections in terms of the inheritance relationship. The third category, type 10 to 12, is for the connections with aggregation and association relationship. Thus there are a total of 12 connection types which capture the connections between the classes of the package. Thus we have class-attribute interaction (aggregation), class-method interaction and method-method interaction [36] plus the interaction relationship which has been represented as a set of connections in our table. Thus we can see here that the relationships are also interactions at the end of the day which is evident from the table we have above.

Association: It represents a general binary relationship that describes an activity between two classes. Relationships between classes (or objects) are called **associations** and denoted by solid line connecting the classes.

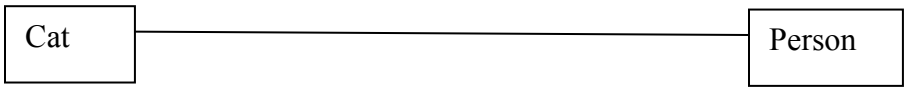

Figure 4.1: Example 1 of Association

Associations can be further specified by assigning roles to the classes or objects involved and naming these roles:

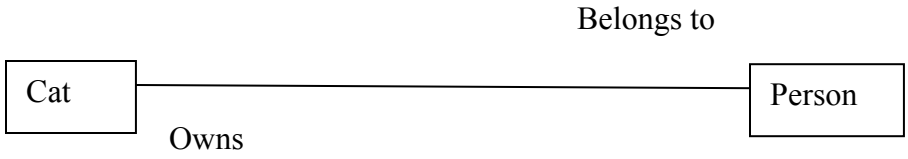

Figure 4.2: Example 2 of Association

Aggregation: It is a special form of association, which represents an ownership relationship between two classes. Aggregation models the relationship like has-a, part-of, owns, and employed-by. Although the parts may exist independently of the whole, their existence is primarily to form the whole. It supports roles and multiplicity (because it is an association). In terms of implementation in java, an aggregation maps to instance variables on a class. In a class diagram, an aggregation is represented by drawing an edge with a large dark circle on the end which contains objects at the other end.

In general, unless you believe that using aggregation adds value or clarifies something, you should use association.

Inheritance: It models the is-a relationship between two classes. It is a relationship between classes where one class is the parent class of another derived class called base class or super class. In deriving a class from a base class, one must understand all the properties of the base class to fully understand the derived class. Use of inheritance can lead to reduction in the amount of code developed to represent similar program structures

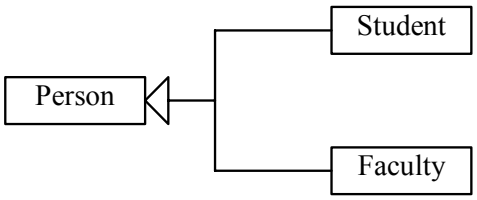

Figure 4.3: Example of Inheritance

4.2 Indirect Connections

Eder *et al*. [16] derive "indirect interaction relationships between methods" from "direct interaction relationships" using the transitive closure of direct interaction relationships. This idea can be applied to all kinds of coupling. If a class *c1* uses a class *c2*, which in turn uses a class *c3*, class *c1* is indirectly coupled to *c3*: a defect or modification in class *c3* may not only affect the directly coupled class *c2*, but also the indirectly coupled class *c1*. As an extreme case, consider a circular chain of coupled classes (class *ci* uses class *ci+1* for *i=1,2,...,n-1*, and class *cn* uses *c1*). Each class is directly coupled with two other classes (import and export coupling). However, each class in the chain indirectly uses and is being used by every other class.

Most of the coupling measures consider direct coupling only. RFC′ is the number of methods that can possibly be invoked by sending a message to a class *c*. This includes methods of *c*, methods invoked by the methods of *c*, the methods these in turn invoke, and so on. In that sense, indirect coupling is accounted for. RFCα counts such nested method invocations up to a specified level α. We have to decide whether to count direct connections only or also indirect connections.

Indirect connections can be relevant when estimating the effort for run-time activities such as testing and debugging, or to estimate the impact of a modification to a class *c* on the system: the modification may necessitate other modifications to classes directly and indirectly connected to class *c* (ripple effects). Indirect connections may also be relevant for reusability: if a class *c* is to be reused in another system, not only the classes to which *c* is coupled have to be provided in the system, but also classes required by these coupled classes.

4.3 Linking Package cohesion principles to the Metrics

Release Reuse Equivalency Principle

THE GRANULE OF REUSE IS THE GRANULE OF RELEASE

The REP states that the granule of reuse can be no smaller than the granule of release. Therefore, one reason for grouping classes into packages is reuse. Since packages are the unit of release, they are also the unit of reuse. Therefore architects would do well to group reusable classes together into packages. The users cannot reuse anything that is not also released. When users reuse something in a released library, he is a client of the entire library. Anything that is reused must also be released. Clearly, packages are a candidate as a releasable entity.

The Unit of Release therefore has a direct relation to the cohesiveness of the packages involved because a package having good Interaction cohesion is definitely going to be a candidate for reuse because of its cohesiveness and integrity which is a desirable feature for any unit to be reused (package) as clearly they are a candidate for releasable entity as explained above. Also the metrics discussed above gives the cohesiveness of a package with respect to the interaction, inheritance and association-aggregation connections involved in the package, which helps in giving a clear understanding of the packages which are desirable to be used as a reusable unit through their package cohesion values because the higher the package cohesion values the higher is the reliability of the package and lower the maintainability of the concerned package.

Common Closure Principle

CLASSES THAT CHANGE TOGETHER, BELONG TOGETHER

When we group classes that change together into the same packages, then the package impact from release to release will be minimized. The CCP is an attempt to gather together in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, such that they almost always change together; then they belong in the same package.

Of course this is a desirable principle as we want to reduce the maintainability costs as much as possible as we tend to confine the propagation of a problem to other packages. Therefore, to achieve this we have to look for the cohesiveness of the package involved at different versions because we need to monitor the changes that classes undergo which may be a part of the requirements or for any other modification, and this can be achieved by looking at the cohesion values of the package through the different metrics discussed above reflecting the interaction, inheritance and association, aggregation connections in the package , as lower the package cohesion values over different version for a package, the higher the probability that the package needs rearrangement of the classes in order to group the classes that are likely to change for the same reasons.

Common Reuse Principle

CLASSES THAT ARE NOT REUSED TOGETHER SHOULD NOT BE GROUPED TOGETHER

This principle helps us to decide which classes should be placed into a package. It states that classes that are not to be reused together should not belong in the same package. If

classes that are not used together are grouped together, changes to a class that is not used about will still force a new release of the package, and to go through the effort of upgrading and revalidating.

This principle reinforces the idea of the previous principle necessitating the use of having a package constituted of classes which are relevant to one another and this too can be achieved by looking at the cohesive values of the package using the metrics presented. Lower values of package cohesion indicates that there is something wrong with the packaging of the classes as the messaging or interaction which can be the normal interaction between the classes or through he use of specialized connections like inheritance and association –aggregation, between the classes of the package is not much which would help us in decide which classes should be placed in a package which in turn increases the relevance of having classes together in a package and thus increases the cohesiveness of the package.

Thus the metrics presented in our work can be used to gauge whether the package under consideration meets the desired guidelines and principles which are necessary in order for the package to do what it is designed for and that is to achieve a cohesive unit working towards realizing a part of the functionality which makes up a system. The metrics presented here gives a thorough check on the constitution of the package's elements with its detailed emphasis on all possible interactions like the normal interaction between two classes accessing a method or attribute to the more specialized connections like the inheritance and the association and aggregation relationships, which are of importance in determining the cohesiveness of the package.

4.4 New Package Cohesion Metric

As we have divided the connection types into three categories: Interaction, Inheritance and Association. We have 3 metrics in total. Three of those represent each of the above mentioned categories.

4.4.1 Package Interaction Cohesion (PIC)

Here we calculate the metric as the no. of methods in a particular class that have a connection (i.e., interaction) with the methods of other classes. This is explained with an example as shown below.

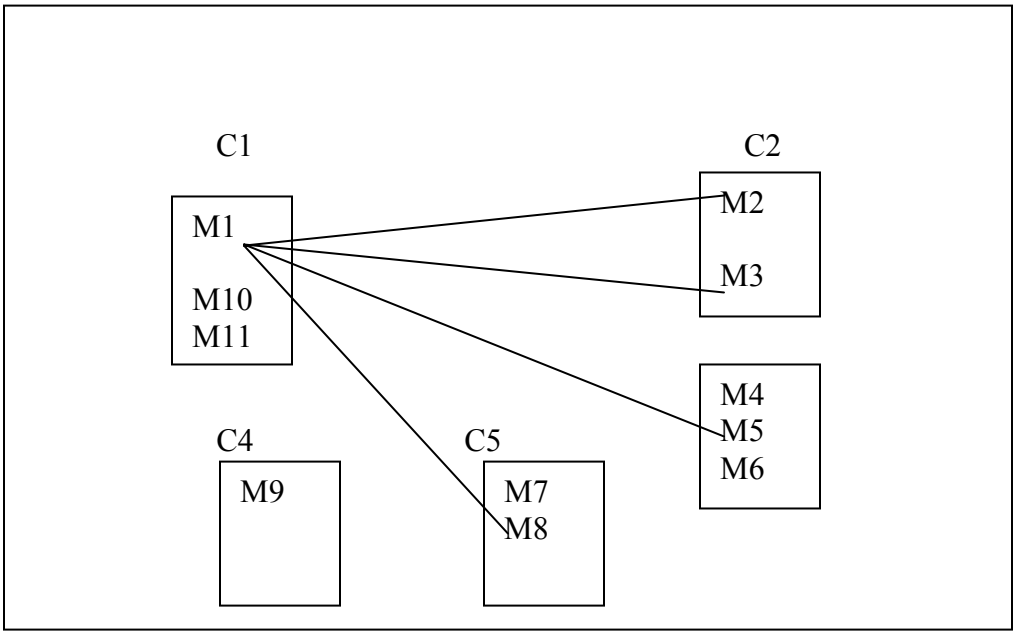

Figure 4.4: Package Interaction Cohesion: Example

For example in the figure above the no. of methods having connection is only one. So we say that,

$$\frac{1 \text{ (No. of methods having connection)} + 2 + 1 + 1 + 0}{3 \text{ (Total no. of methods in the class)} + 2 + 3 + 2 + 1} = 0.45$$

This way we calculate the connections of the methods in each class to every other class in the package. And then sum all these results and divide it over the total no. of methods in all the classes of the package under consideration. In an equation form

$$\frac{\sum_{i}\sum_{ci}(m)}{\sum_{i} m(ci)}$$

Where, the numerator is the sum of connections of methods for a particular class which includes the connection numbers from 1 to 5 which have been highlighted in the table above corresponding to each connection type. The next summation exemplifies the same thing over all classes. The denominator represents the total no. of methods in all classes.

4.4.2 Package Inheritance cohesion

The connection types representing the package Inheritance cohesion metric are listed in the table 4.1. The connection types 6 to 9 correspond to this metric. We calculate this metric using the same formula as the package Interaction cohesion metric which is depicted below.

$$\frac{\sum_{i}\sum_{ci}(m)}{\sum_{i} m(ci)}$$

Where, the numerator is the sum of connections of methods for a particular class which includes connections 6 to 9 corresponding to the possible Inheritance connection types, having a connection with other classes. The second connection does the same thing for each class in the package under measurement.4.4.3 Package Association cohesion

The Package Association connection types are listed in the table from 10 to 12. These connection types also include the Aggregation connection type thus completing a comprehensive list of connection types related to the package cohesion measurement.

$$\frac{\sum_{i}\sum_{ci}(m)}{\sum_{i} m(ci)}$$

The above formula illustrates how we calculate the Package association cohesion. Again the formula here is the same barring, of course, the connection types being scrutinized here. For each class we look at the connections (connection types 10, 11 & 12) it has with every other class and then the sum is divided over with the total no. of methods in all the classes. The results of various projects ran for the above mentioned metrics are elaborate in chapter 5.

4.5 Theoretical Validation

Several researchers have proposed properties that software metrics should posses in order to increase their level of confidence. It is desirable to have a formal set of criteria with which to evaluate proposed metrics.

In 1996, Hitz and Montazeri [20] used the concept of measurement theory to evaluate and validate any given metric. They identified the significance of establishing a "sufficient" empirical relation system after the researcher has identified his attributes of interest. Having established an empirical relation system, a metric M should then map the empirical relation system into an appropriate formal (or numerical) relation system, preserving the semantics of the empirical relation(s) observed. In other words, for every empirical relation $\angle$ and a corresponding formal relation <, the representation condition

$X \angle Y \Leftrightarrow M(X) < M(Y)$ must hold. The task of validating a software measure in the assessment sense is equivalent to demonstrating empirically that the representation condition is satisfied for the attribute being measured [37].

So, the empirical relation will be stated as: *The more edges in the interaction graph,G_X, the higher the cohesion of the class X in other words it should satisfy monotonicity.* Therefore any metric M should preserve the semantic of empirical relation. In 1998, Briand et al. have proposed a mathematical framework including properties to be satisfied by several types of software metrics [36]. Cohesion measure is one of the measures supported by this framework, others include: size, length, coupling and complexity. The following properties are proposed with respect to cohesion metrics, in other words, any well define metric should satisfy the following conditions.

Property 1: Non-negativity and Normalization

The cohesion of a class of an object oriented system should belong to a specified interval (i.e. Cohesion (C) ε [0, Max]. Normalization allows meaningful comparisons between the cohesions of different classes, since they all belong to the same interval.

Property 2: Null value and maximum value

The cohesion of a class of an object oriented system is null if there is no interactions among the components of the class (i.e. interaction among the methods and attributes of the class) and it is maximum if the interaction among the components is maximal.

Property 3: Monotonicity

Let C be an object-oriented system, and $c \in C$ be a class in C. Assuming we modified the class c to form a new class c' which is identical to c except that there are fewer

interactions in c than in c'. Let C' be the object-oriented system which is identical to C except that class c is replaced by class c'. Then

$$[cohesion(c) \leq Cohesion(c') \mid Cohesion(C) \leq Cohesion(C')]$$

In order words, if a relationship is added to an object-oriented system, cohesion must not decrease.

Property 4: Merging of unconnected classes

Let C be an object-oriented system, and $c_1, c_2 \in C$ be two classes in c. Let c' be the class which is the union of c_1 and c_2. Let C' be the object-oriented system which is identical to C except that classes c_1 and c_2 are replaced by c'. If no relationship exist between classes c_1 and c_2 in C, then

$$[\max\{Cohesion(c_1), Cohesion(c_2)\} \geq Cohesion(c') \mid Cohesion(C) \geq Cohesion(C')]$$

In other words, the merging of two unconnected classes must not increase cohesion (because the union of two unconnected classes will have little cohesion).

Hermadi et al. augmented two other additional properties to Briand's et. al. framework, that cohesion metrics need to satisfy; these are symmetry and transitive [38]. These properties are defined below.

Symmetry: the cohesion of a class should not be sensitive to the direction of the relation between its components. If there is a relation between m1 and m_2 then the representation of $m_1 \rightarrow m_2$ is equivalent to $m_2 \rightarrow m_1$.

Transitivity: Consider three classes c_1, c_2 and c_3 such that, Cohesion (c_1) < Cohesion(c_2) and Cohesion(c_2) < Cohesion(c_3), then Cohesion(c_1) < Cohesion(c_3).

Chidamber and Kemerer [18] proposed in their framework the following four properties: The Theoretical Validation of Giancarlo Metric is as follows:

Non-negativity and normalization: CC cannot be negative because it is the sum of the number of internal classes to which a class is coupled. This value has a defined maximum equal to 1, reached when all the classes are coupled among them-selves.

Null value: CC is zero if the classes inside our component are not coupled.

Monotonicity: If we add internal relations inside a component, obviously the CC does not decrease, but it either increases or remains the same.

Cohesive modules: If two modules that do not have relations between them are merged, CC does not increase. In fact, the number of relations among the classes inside our component will be the same

4.5.1 Theoretical validation for our metric.

We here validate our metric against the union of the properties that have been found in the literature. All the authors Briand, Hitz, Hermadi have the following 5 properties in their theoretical validation of their metric, and thus we too will validate our metric against these five proerties.

- Non-negativity

Our Package cohesion metric cannot be negative because it is the sum of the number of internal classes to which a class is connected. Therefore our metric satisfies this property

$$\text{PIC} = \frac{\sum_{i}\sum_{ci}(m)}{\sum_{i} m(ci)}$$

- Null value and maximum value

The metric has a null value when there is no connections between the classes , so the numerator is zero and hence the metric value will be zero, on the other hand if the classes

are fully connected then the numerator and denominator will evaluate to 1 which is the maximum value. Hence the metric satisfies this property too

- Normalization

The metric also satisfies the normalization property as the value lies between 0 and 1. If the package cohesion value = 0 then there is no interaction among the classes and hence the value of the numerator, which considers the sum of connected methods will be zero, thus making the metric value zero and it is = 1 if all methods are directly or indirectly connected (i.e. if the interactions between the classes is maximal), as can also be seen from the above Null & Maximum value property. Therefore, the package cohesion value lies in the interval [0, 1] inclusive.

- Monotonicity

Let C be an object-oriented system, and c $\in$ C be a class in C. Assuming we modified the class c to form a new class c' which is identical to c except that there are fewer interactions in c than in c'. Let C' be the object-oriented system which is identical to C except that class c is replaced by class c'. Then,

$$[cohesion(c) \leq Cohesion(c') \mid Cohesion(C) \leq Cohesion(C')]$$

So for our metric, PIC= $\frac{\sum_{i}\sum_{ci}(m)}{\sum_{i} m(ci)}$, the numerator is the count of the summation of

interactions between the methods of the classes under consideration, and if we remove connections then the count of the connections in the numerator will decrease, and since the denominator which is the number of methods in the classes remains the same, obviously the cohesion value will decrease, for this fraction. Similar is the case when we

add connections, the count of the connections in the numerator will increase and so will the cohesion value. Let us visualize this with the help of an example.

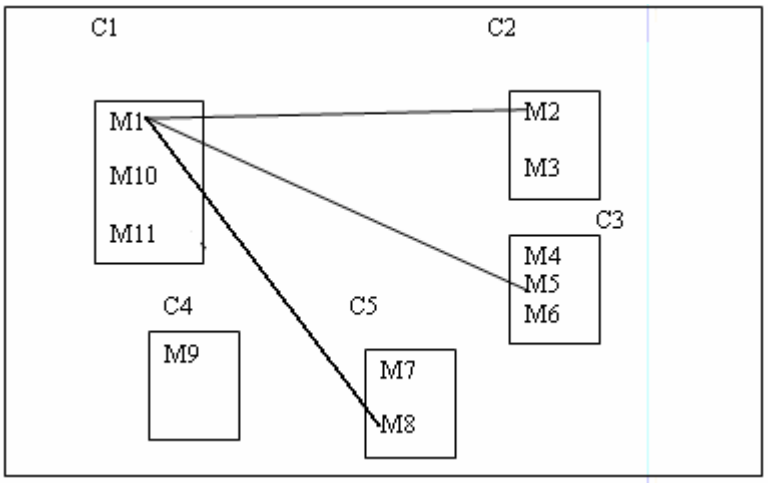

Figure 4.5: Monotonicity Example

The equation for C1 will now be

$$\frac{1 \text{ (No. of methods having connection) } +1 +1 + 1 +0}{3 \text{ (Total no. of methods in the class) } + 2 +3 +2 +1} = 0.36$$

The value is less than what we got before (Figure 4.4), i.e., after we reduced a connection from m1 to m3. So the condition for monotonicity, which states that the reduction of connection should reduce the cohesion, is true for this case. Similar is the case vice versa.

- Merging of unconnected classes

If two classes that do not have relations between them are merged, the cohesion value does not increase. In fact, the number of relations among the classes inside our component will be the same.

$$\frac{\sum_{i}\sum_{ci}(m)}{\sum_{i} m(ci)}$$

As is evident from the above equation, if you add methods that do not have a connection, then the cohesion value will decrease, as the metric calculates the number of connections between methods in the numerator and since there are no newly added connections the numerator remains the same and correspondingly the denominator will increase because of the addition of methods, as the denominator will count the number of methods and this will decrease the cohesion value.

Chapter 5

Results and Discussion

5.1 Interaction Package cohesion values

The following results depict the Package interaction values (PIC) for the various systems that have been tested. For example Jext has 41 packages and the system showed varied results for each of its packages ranging from as low as 0.1 to as high as 0.9. The lower values however should ring a bell to the designer and he needs to have a deeper look at the particularly low PIC valued packages. The X-axis shows the no. of packages in each system and the Y-axis shows the normalized metric values.

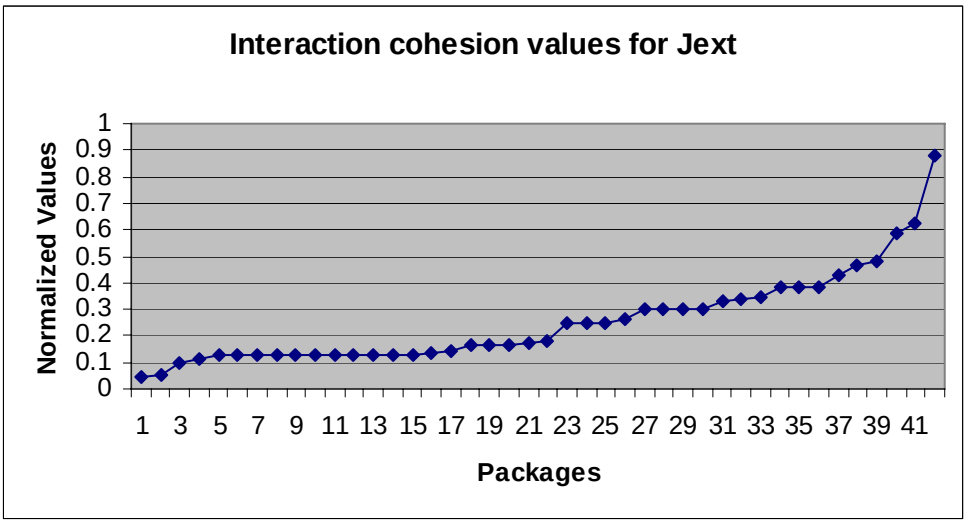

Figure 5.1: Interaction Cohesion Values for Jext

Another system called Saxon was tested for PIC values and it showed varied PIC values. However the values were lesser than Jext. The system saxon 6.5.2 showed a maximum value of 0.5 , it had 23 packages.

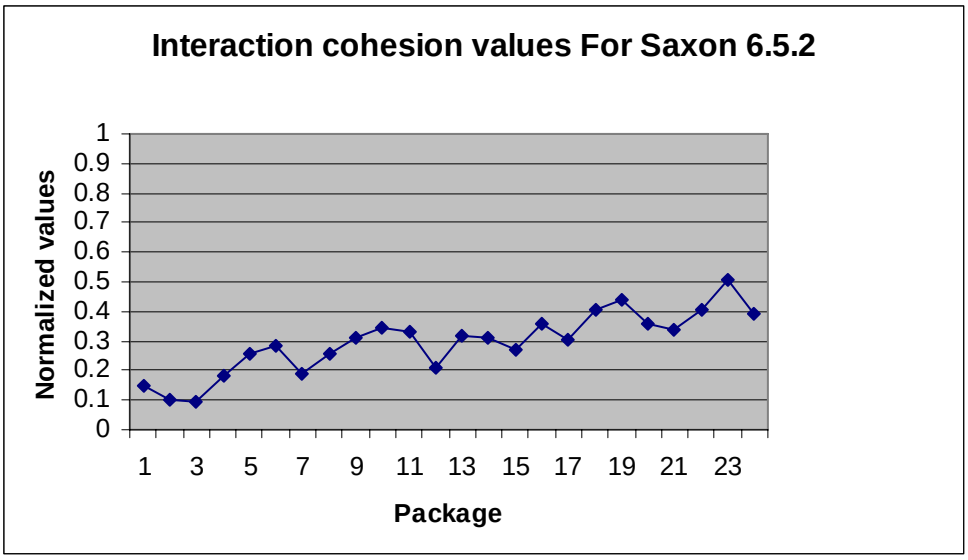

Figure 5.2: Interaction Cohesion Values for Saxon 6.5.2

To see whether these values are improved in the later version we ran the experiment on another version Saxon 8.0 .The results showed that the PIC values were better in the following version when compared to its predecessor, the maximum PIC value for some packages to be as high as 0.6 when compared to its predecessor's high of 0.5.

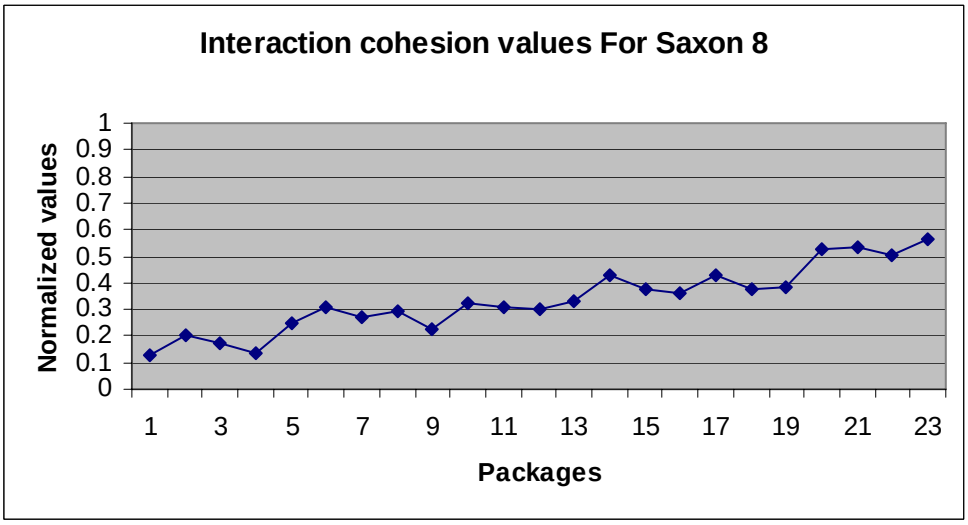

Figure 5.3: Interaction Cohesion Values for Saxon 8

Babeldoc 1.0 was tested, which showed good cohesion values with the maximum value of 0.8. Figure 5.4 depicts it.

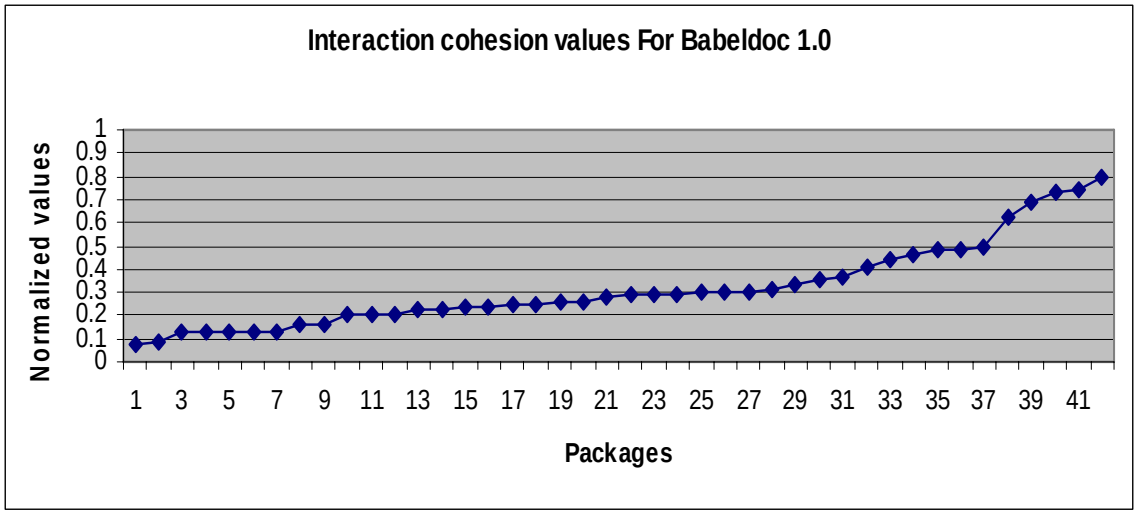

Figure 5.4: Interaction Cohesion Values for Babeldoc 1.0

5.1.1 Effect of Indirect Connections

We ran experiments to see the effect of not only direct connections but also indirect connections. For example, if a method *m1* invokes a method *m2*, which in turn invokes a method *m3*, we can say that *m1* indirectly invokes *m3*. Methods *m1* and *m3* are indirectly connected. Indirect connections can be relevant to estimate the impact of a modification to a class *c* on the system: the modification may necessitate other modifications to classes directly and indirectly connected to class *c* (ripple effects). Indirect connections may be relevant for reusability. If a class *c* is to be reused in another system, not only the classes to which *c* is directly coupled with have to be provided in the system, but also classes required by these coupled classes and so on.

Therefore we ran the experiments on the same systems to see the effect of indirect connections. The results were in accordance with hypothesis of Briand [36] that the

values of indirect connections should be equal to or greater than the direct connections. And this is clear in the graphs seen below that almost always the indirect connections are equal to or greater than the direct connections.

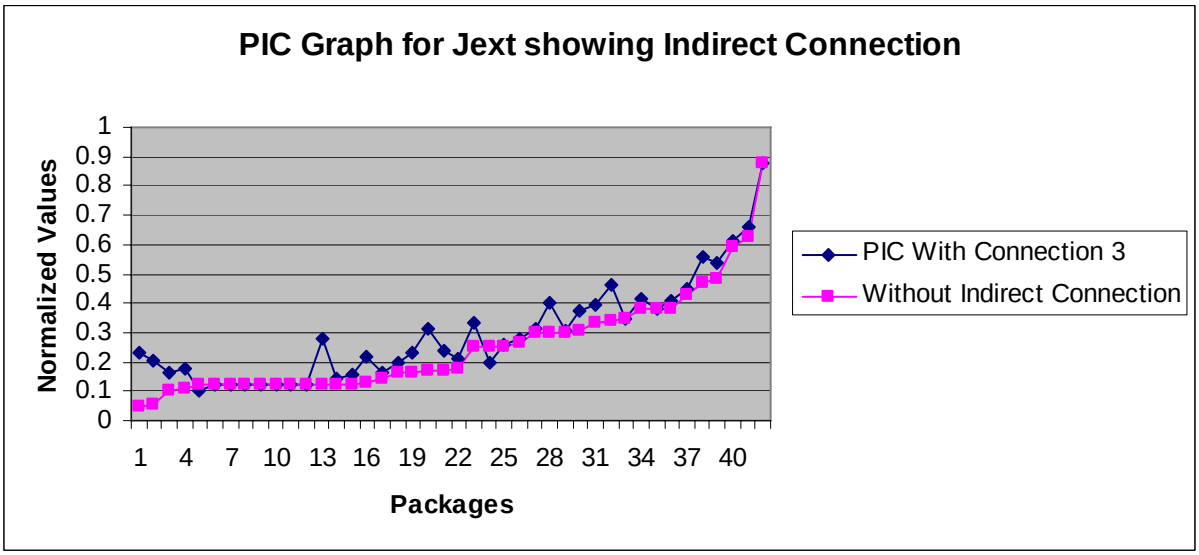

Figure 5.5: PIC Graph for Jext Showing Indirect Connection

All the systems showed similar trends i.e., the indirect connection values were at least equal to that of their counterpart and in some cases bettering the values.

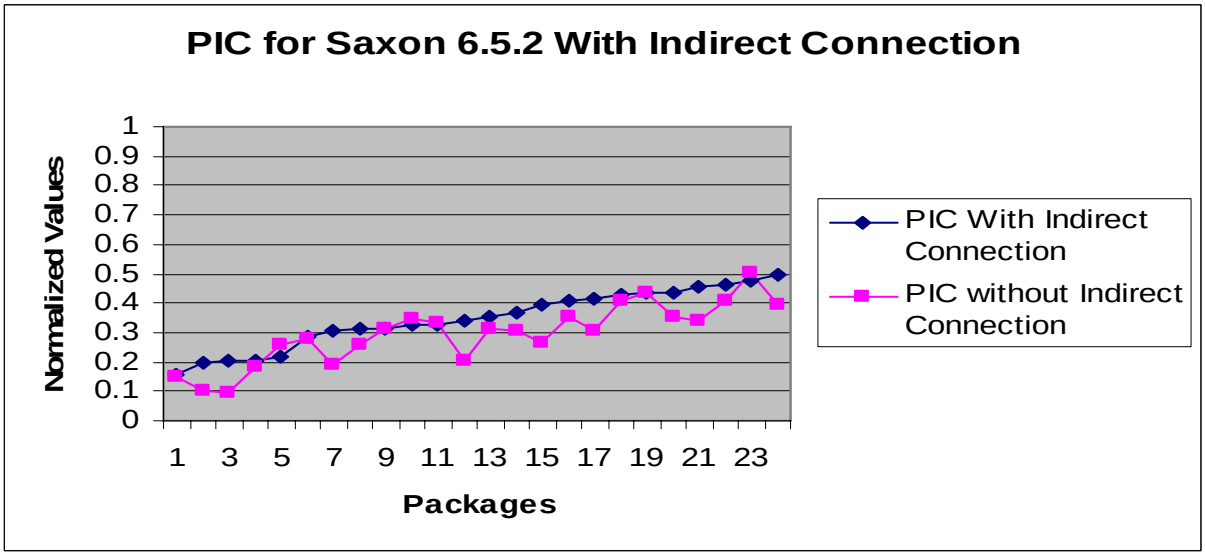

Figure 5.6: PIC for Saxon 6.5.2 with Indirect Connection

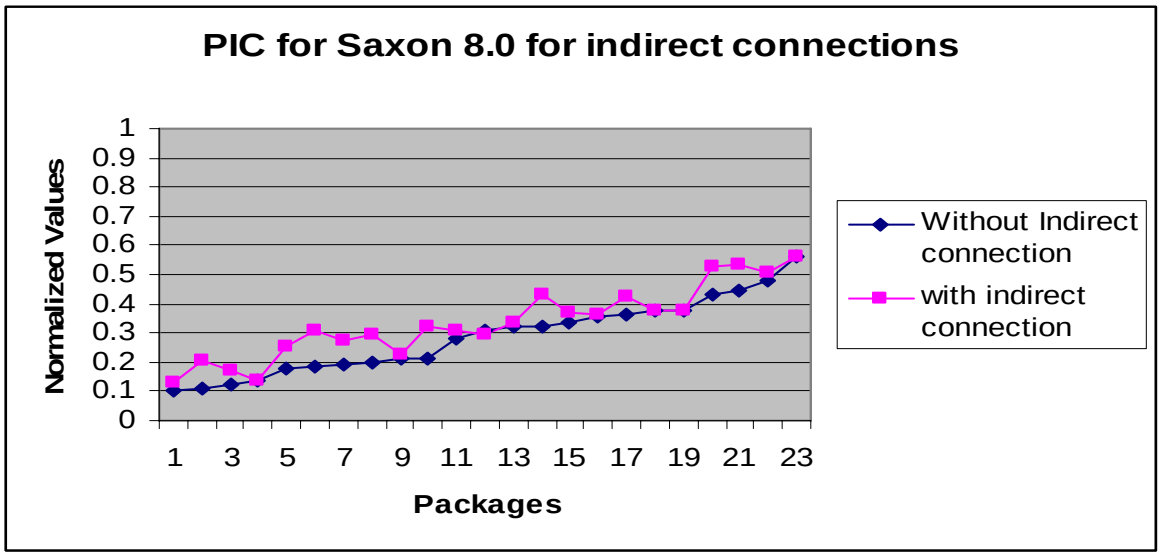

Figure 5.7: PIC for Saxon 8.0 with Indirect Connection

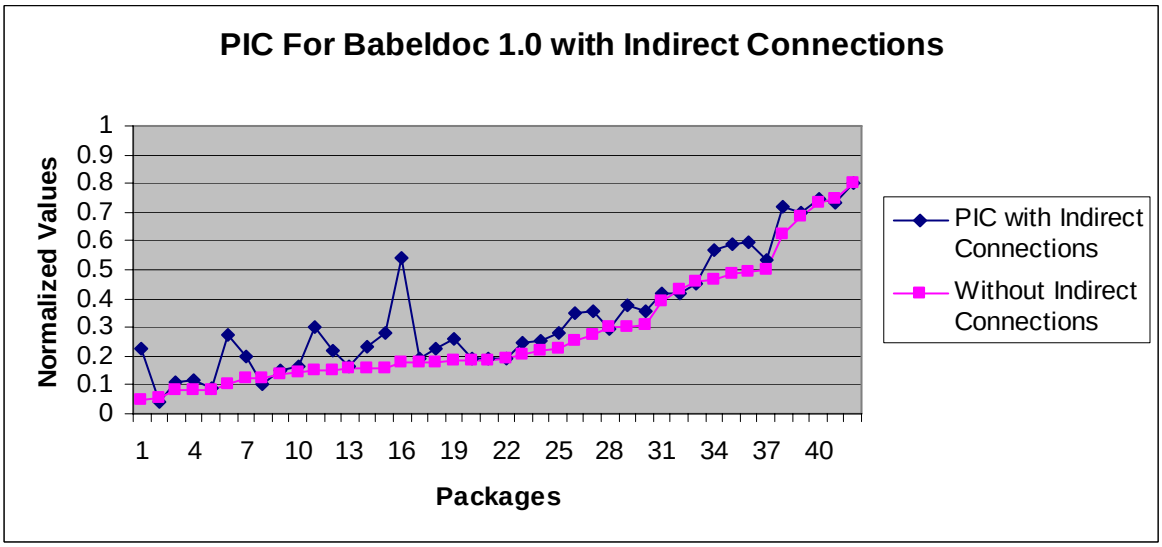

Figure 5.8: PIC for Babeldoc 1.0 with Indirect Connection

5.2 Package Inheritance Cohesion (PInC)

We ran the experiments to see the effect of inheritance on packages. The results for Jext, Saxon 6.5.2 and Saxon 8.0 are as follows. We found that most of the systems where having PInC values as Zero, this may be due to the fact that either the concepts of

Inheritance was not being used often or being used incorrectly i.e., unnecessarily inheriting the classes which are not being used later. The maximum PInC values seen are around 0.4

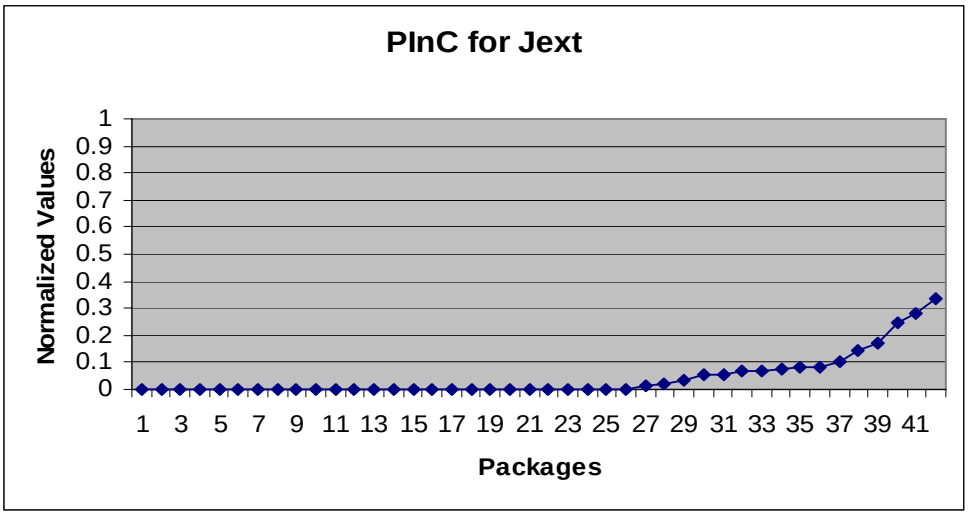

Figure 5.9: Package Inheritance Cohesion for Jext

As can be seen in the Figures 5.10 & 5.11 the Package inheritance values for the two versions of Saxon are low but still the later version of Saxon i.e., Saxon 8.0 shows improvement over its predecessor with the maximum value reaching nearly 0.4

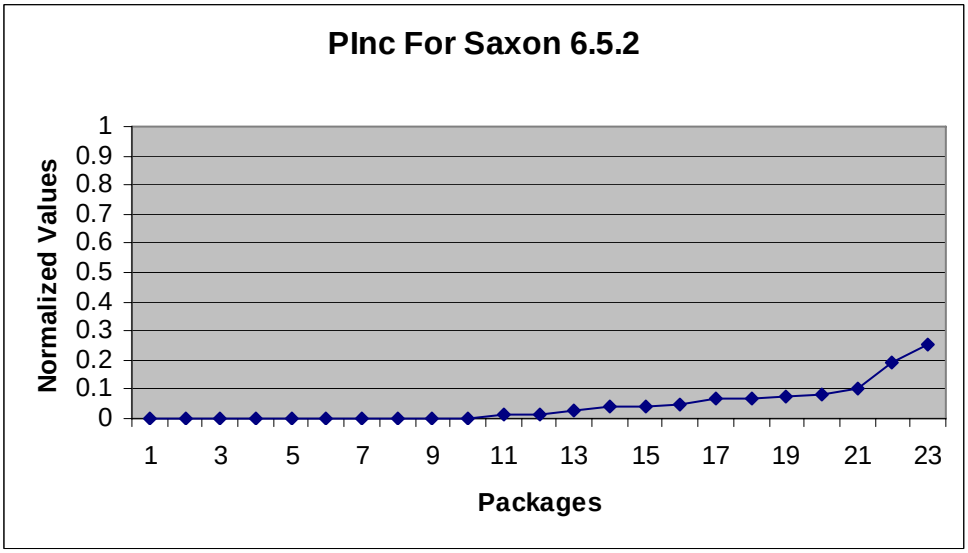

Figure 5.10: Package Inheritance Cohesion for Saxon 6.5.2

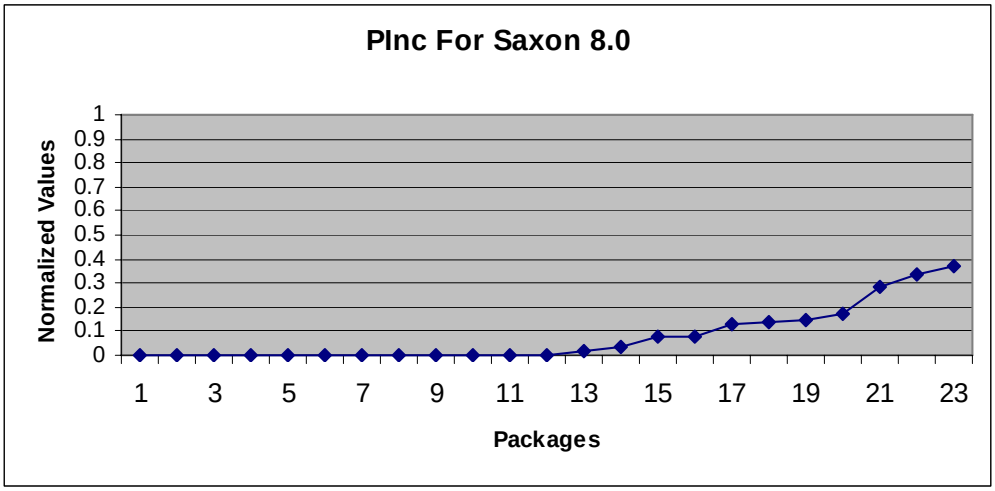

Figure 5.11: Package Inheritance Cohesion for Saxon 8.0

5.3 Package Association cohesion (PAC)

We also tested the above mentioned system to look at what effect association and aggregation connection types have on these systems. We found that most of the systems where having PAC values as Zero, this is primarily due to the fact that these concepts of Association and Aggregation is not been used often and the maximum PAC values seen are around 0.5 to 0.6 in almost all of the systems except Babeldoc 1.0 which had the maximum PAC value as 0.3. These can be seen in the figures to follow.

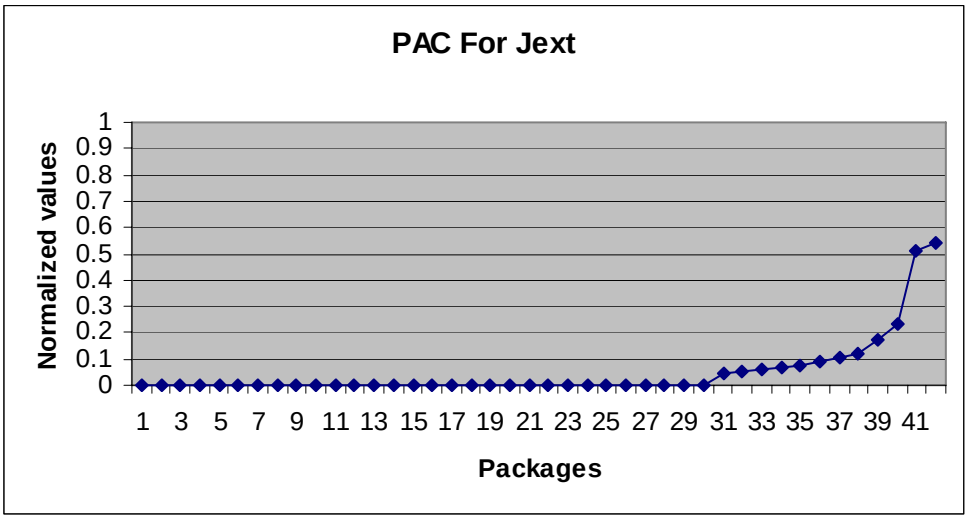

Figure 5.12: Package Association Cohesion for Jext

A point to be noted here is that the Package Association values that we got are better than inheritance cohesion values, which gives an indication of the relationships i.e., Association and Aggregation are being used more often than inheritance relationship. Here also as in the case of Inheritance cohesion the Association cohesion values for the later version of Saxon seem to be improving than that of its predecessor.

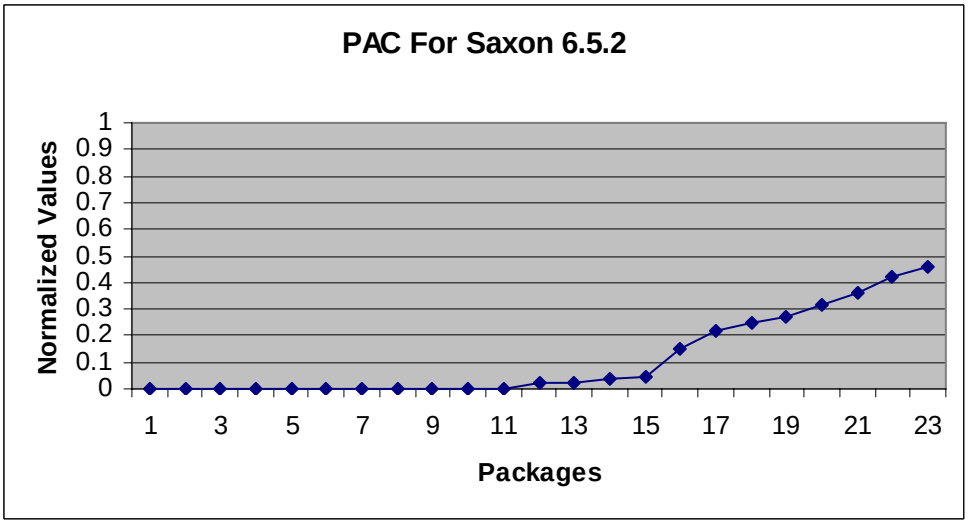

Figure 5.13: Package Association Cohesion for Saxon 6.5.2

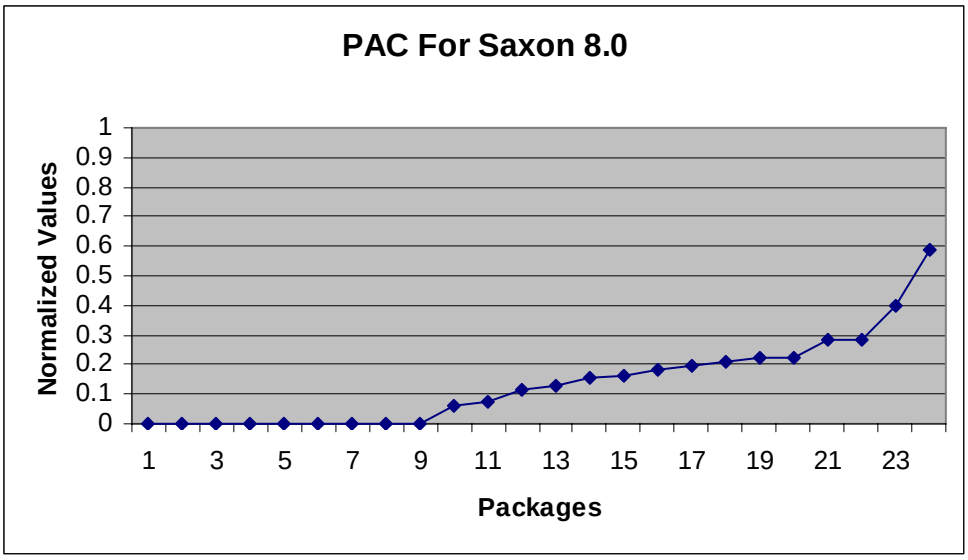

Figure 5.14: Package Association Cohesion for Saxon 8.0

The PAC values for Babeldoc also have many zero values. The maximum PAC value obtained is 0.3. Figure 5.15 depicts the behavior discussed.

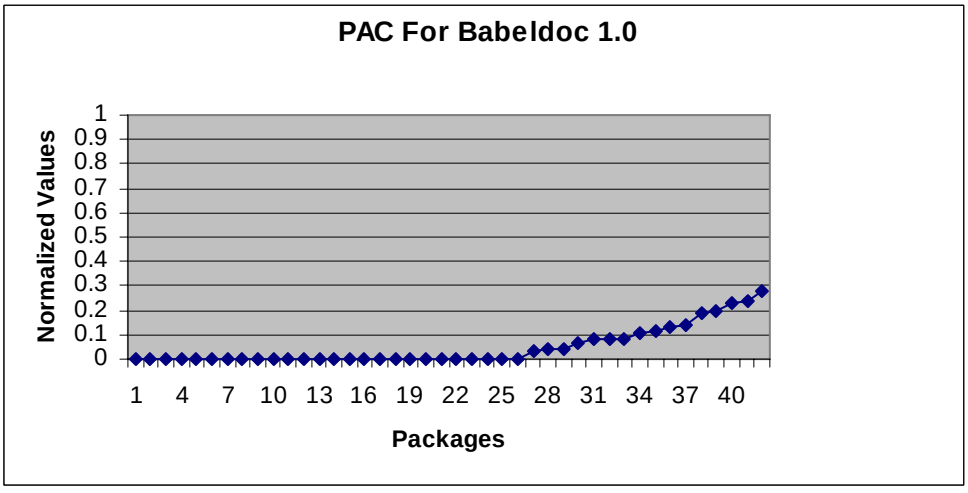

Figure 5.15: Package Association Cohesion for Jext

5.4 Metric results with respect to Jdk versions

In this section we study the well known systems to give more concrete proof of the validity of our metric, as will be seen in the later sections (5.6 & 5.7). The following subsections give the experimental results for different versions of Jdk i.e., Jdk1.2.2, Jdk1.3, jdk1.4. It must be noted that the tests have been carried out on the swing feature of the JDK software.

5.4.1 Jdk1.2.2

To test the performance of our metric with respect to renowned systems which are well known for their cohesion we ran our metric on the different versions of Jdk i.e., on Jdk 1.2.2 , Jdk1.3 and Jdk1.4 and we have noticed that the systems really do show cohesiveness with respect to their interaction as can be seen in the following graphs. They also show use of Association relationships but however the values of inheritance and association relationships have maximum value of 0.4 for JDK1.2.2. The other versions of Jdk like Jdk1.4 show good cohesion values when compared to the Association and Inheritance relationships of their previous versions. Moreover they seem to be increasing with the newer versions like Jdk 1.3 and Jdk1.4, so is the case with the interaction cohesion values for the different versions.

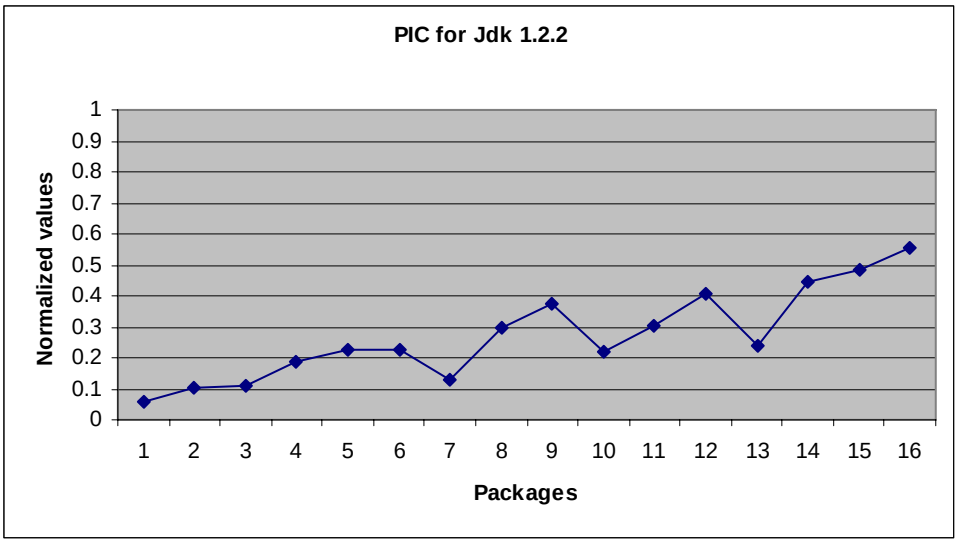

Figure 5.16: Package Interaction Cohesion for JDK 1.2.2

The Figure 5.16 shows the package interaction cohesion values for Jdk1.2.2. There are a total of 16 packages and the curve shows that the maximum interaction cohesion value that has been obtained is nearly 0.6.

The graph below (Figure 5.17) has two curves; one is for the package interaction cohesion which is compared to the second curve, interaction cohesion with indirect connection. The graph confirms that the interaction connections with the indirect connections equal to or as in most cases greater than that of the one without indirect connections.

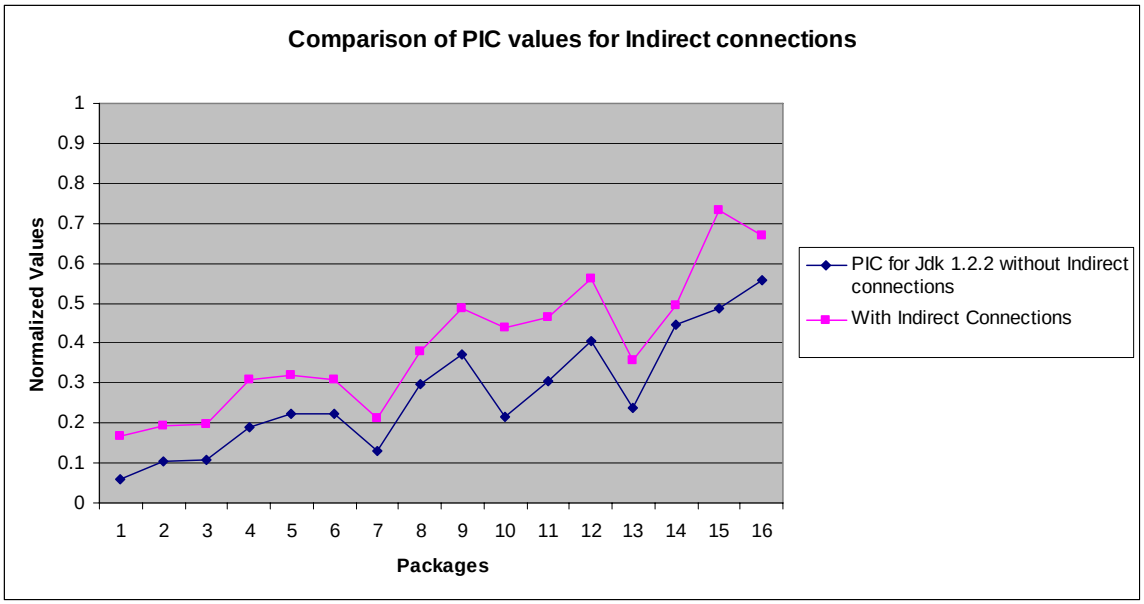

Figure 5.17: Comparison Graph for JDK 1.2.2 with both Direct and Indirect Connections

Figure 5.18 shows the Package inheritance cohesion values for Jdk1.2.2 as can be clearly seen from the graph the maximum cohesion value obtained is nearly 0.3 and most of the packages have zero value which shows that inheritance has been rarely used in the system.

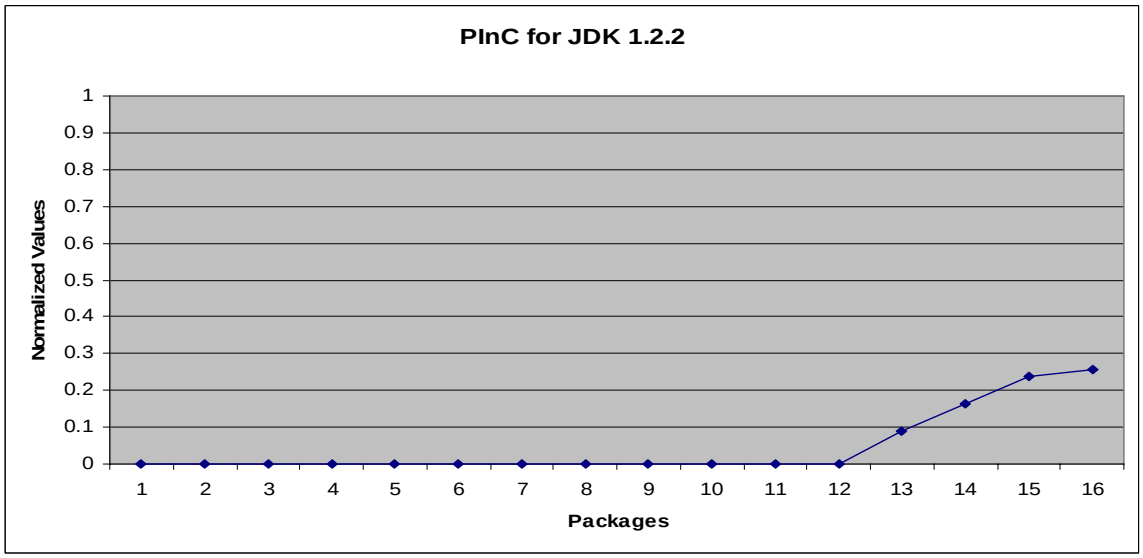

Figure 5.18: Package Inheritance Cohesion for JDK 1.2.2

Figure 5.19 shows the Package Association cohesion values for Jdk1.2.2 as can be seen from the graph the maximum cohesion value obtained is 0.4.

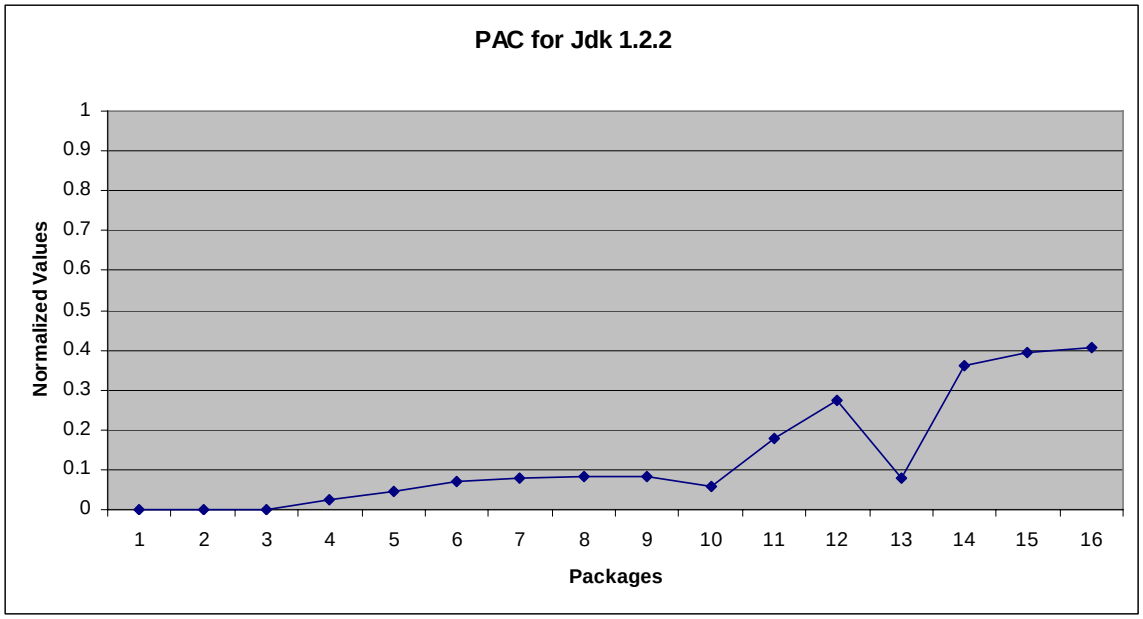

Figure 5.19: Package Association Cohesion for JDK 1.2.2

5.4.2 Jdk1.3

To compare the cohesion values for the different versions of the same software we ran the experiments to see whether there is an increase in the cohesion values as the release increases and as can be clearly seen the cohesion values for some packages are as high as 0.7 and in most of the packages the cohesion values are greater than the previous version i.e., Jdk1.2.2, so is the case for inheritance and association relationships In this version i.e., Jdk 1.3 the no. of packages has increased to 22.

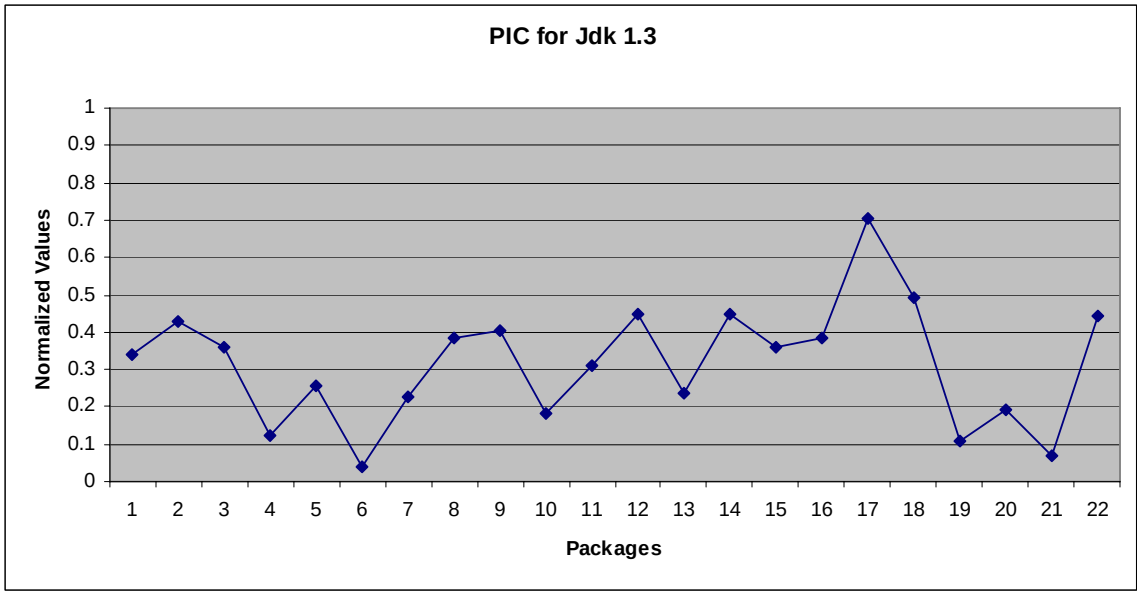

Figure 5.20: Package Interaction Cohesion for JDK 1.3

Figure 5.21 shows the behavior of Jdk1.3 for interaction cohesion values with respect to the indirect connections.

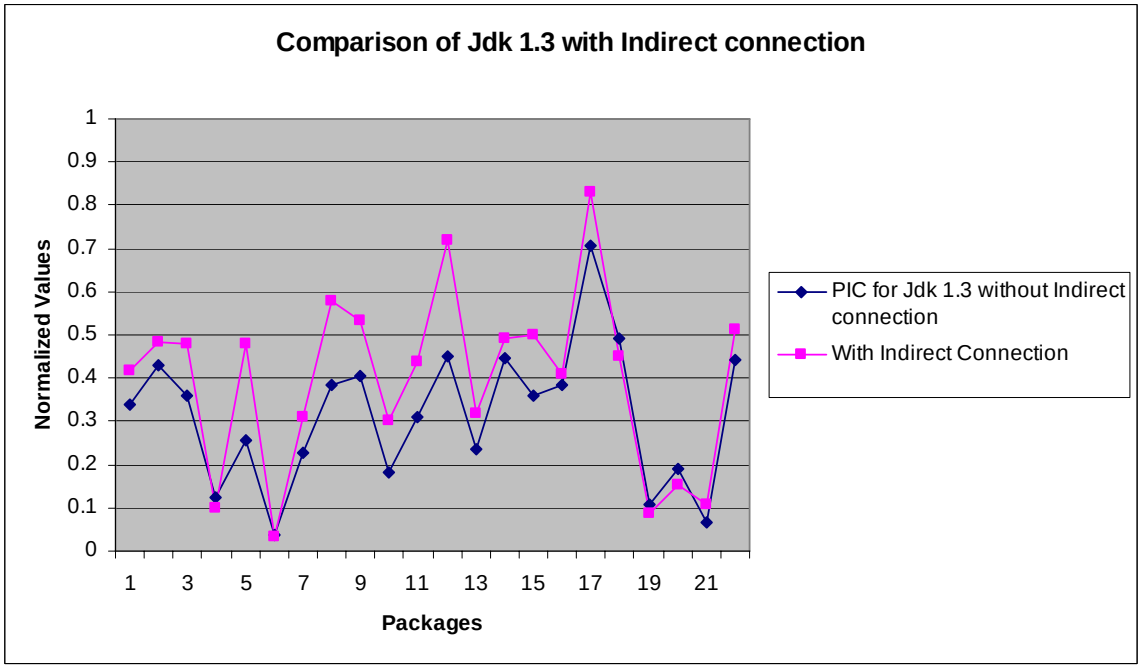

Figure 5.21: Comparison Graph for JDK 1.3 with both Direct and Indirect Connections

The Package inheritance cohesion values although greater than the previous version are still low, with the highest value to be nearly 0.4

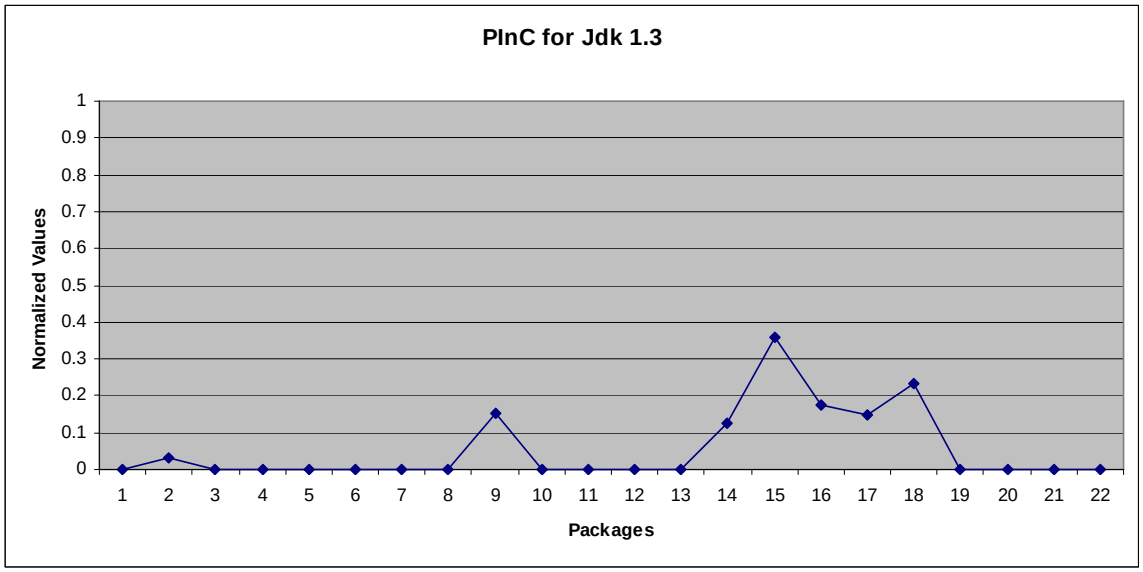

Figure 5.22: Package Inheritance Cohesion for JDK 1.3

The PAC values for Jdk1.3 shows the same trend as Inheritance cohesion values i.e., they increase with respect to their predecessor version but are still low.

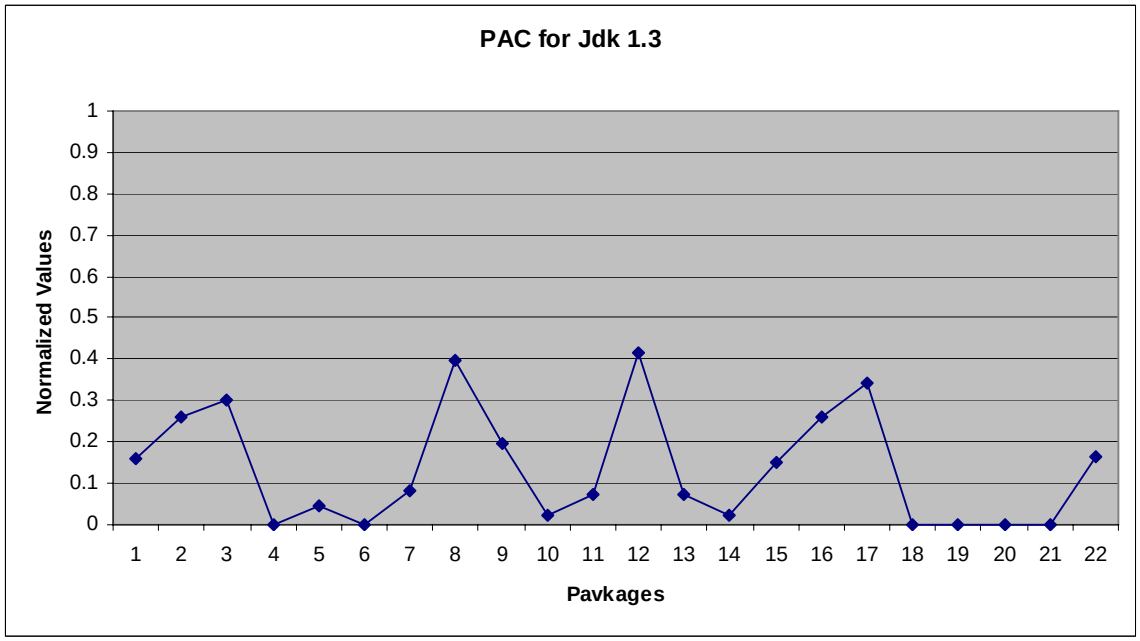

Figure 5.23: Package Association Cohesion for JDK 1.3

5.4.3 Jdk1.4

Similarly as expected the cohesion values for all the three categories of cohesion under consideration seems to be significantly improved when compared with the previous version. For example the maximum Interaction cohesion value for Jdk1.4 is 0.83 which when compared to the maximum cohesion value for Jdk1.2.2 was 0.55. Similarly the maximum inheritance cohesion value for jdk1.2.2 was 0.25 and for Jdk1.4 it is nearly 0.4. Lastly the Association cohesion value for Jdk1.2.2 was 0.4 which when compared to Jdk1.4 is nearly 0.6, the same trend is true for Jdk1.3. Thus it can be confirmed that the cohesion for different versions of Jdk seems to considerably improve and the system exhibits good degree of cohesiveness.

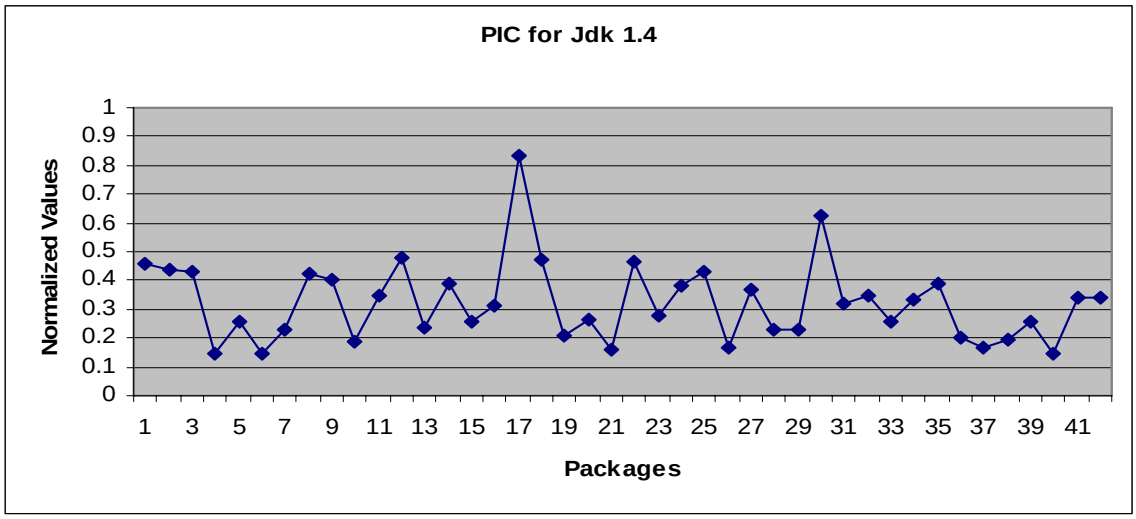

Figure 5.24: Package Interaction Cohesion for JDK 1.4

Figure 5.25 shows the behavior of Jdk1.4 for interaction cohesion values with respect to the indirect connections.

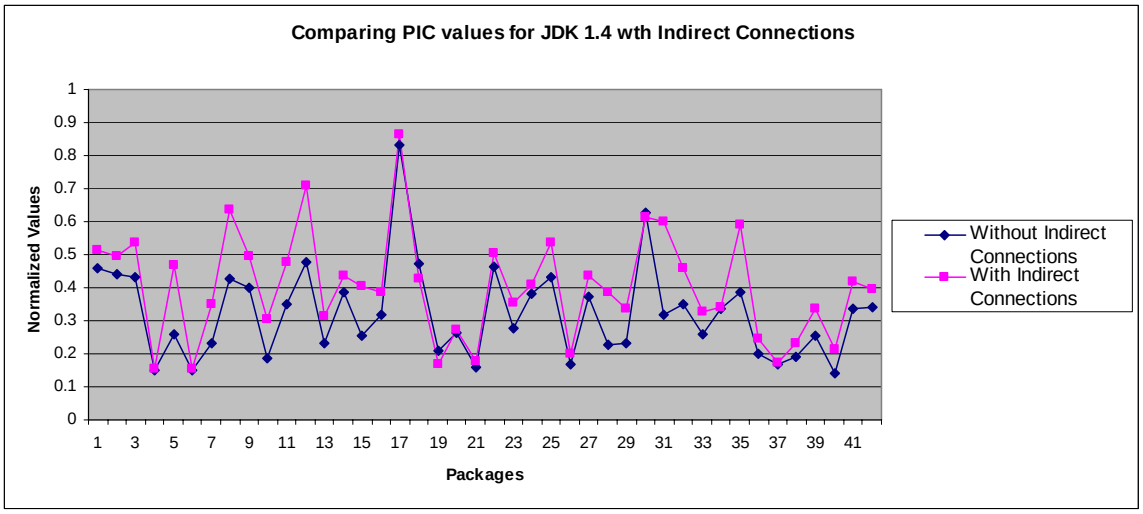

Figure 5.25: Comparison Graph for JDK 1.4 with both Direct and Indirect Connections

The PInC value seems to increase considerably when compared to the previous version especially when compared to Jdk1.2.2 which is a positive sign. Figure 5.26 shows the trend depicted by Jdk1.4 inheritance cohesion values.

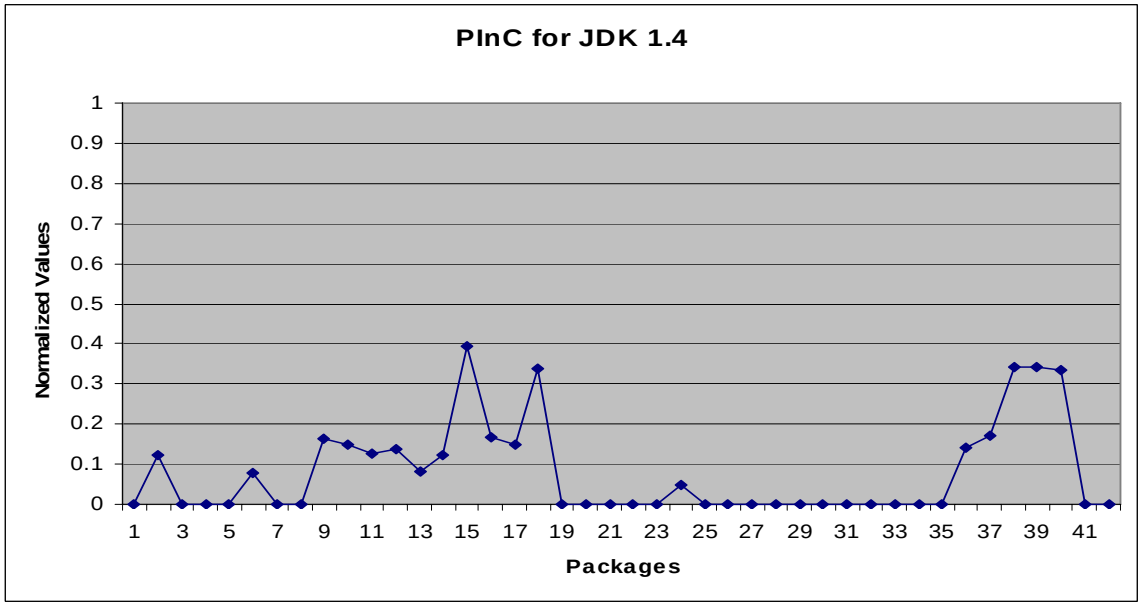

Figure 5.26: Package Inheritance Cohesion for JDK 1.4

Similar is the case for PAC values for Jdk1.4 when compared to its previous versions. Figure 5.27 depicts it.

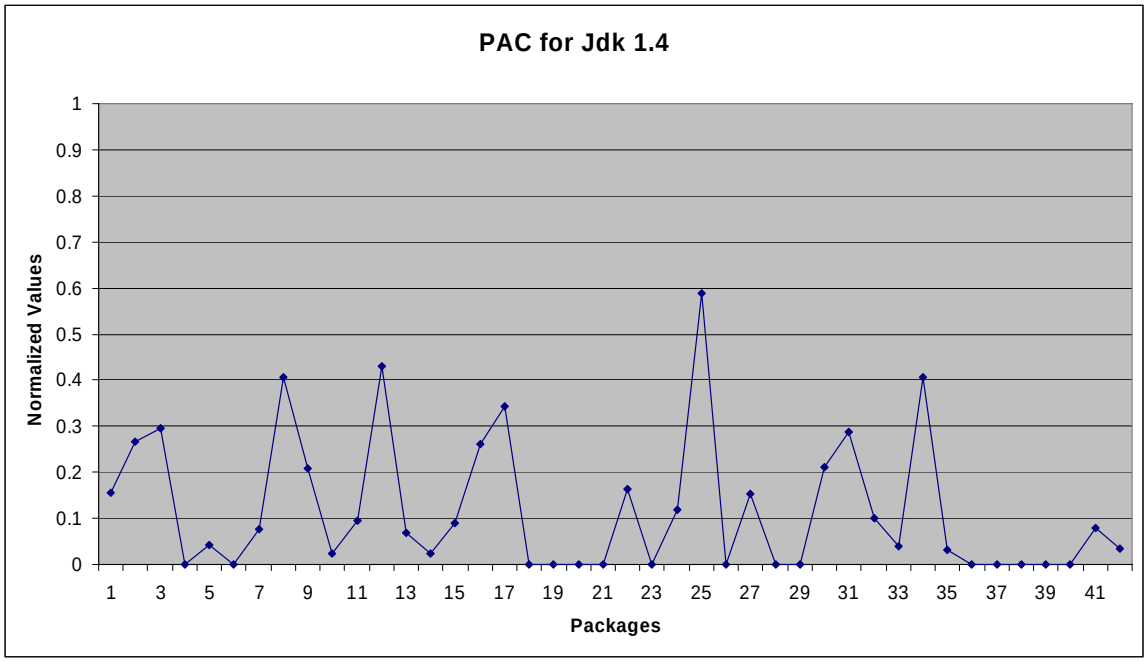

Figure 5.27: Package Association Cohesion for JDK 1.4

5.4.4 Comparison of Jdk versions

In this section we look at the different versions of Jdk to get a deeper insight on the behavior of our metric with respect to the package cohesion values obtained. As can be seen from the graphs outlined below, we have comparisons for the Interaction cohesion metric including indirect connections. We have also Inheritance cohesion comparison and Association-aggregation cohesion comparison values for 1.2.2, 1.3 & 1.4 versions of Jdk swing feature. As expected the package cohesion values for the corresponding versions of Jdk increases for almost all of the packages when compared to their predecessor versions. However it can be noted that package cohesion values of three packages namely, javax.swing.border, javax.swing.undo and javax.swing.tree (which correspond to package no's 14, 15, 16 in the graphs) were decreasing from Jdk 1.2.2 to 1.3 & subsequently 1.4

and this trend was true for The interaction cohesion metric and to some extent for the inheritance and association metric categories.

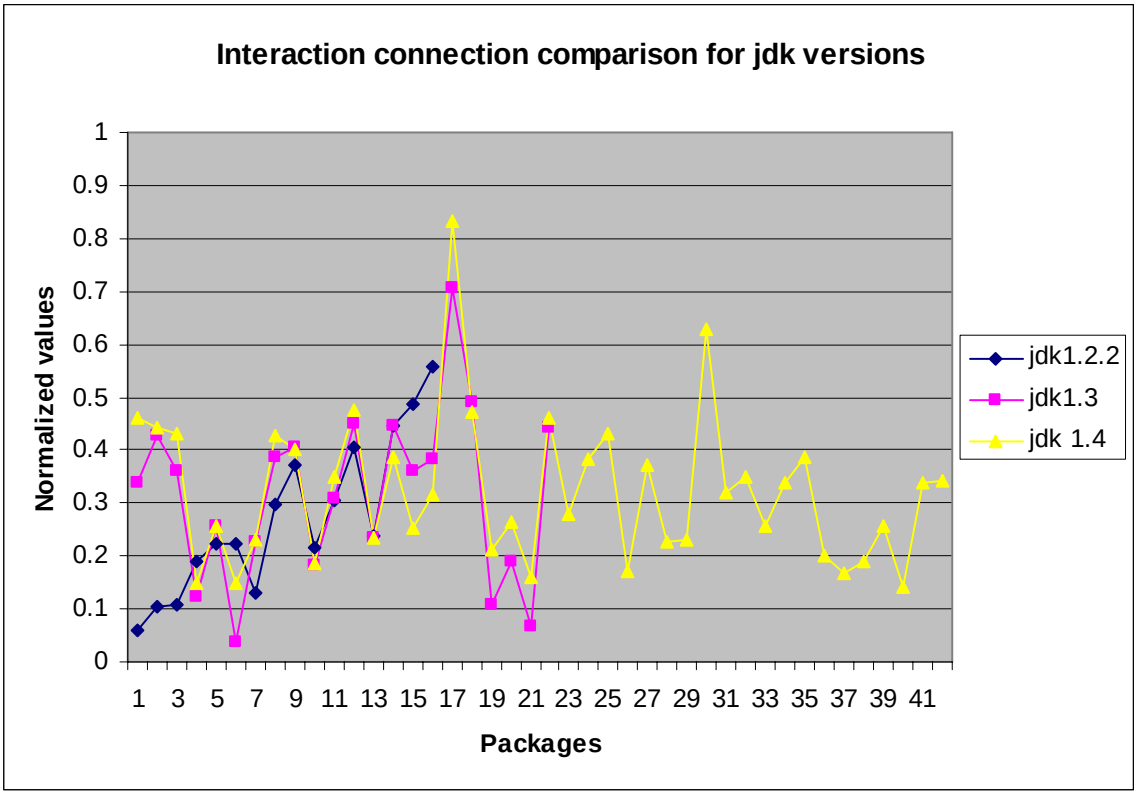

Figure 5.28: Package Interaction Cohesion Comparison Graph for JDK versions

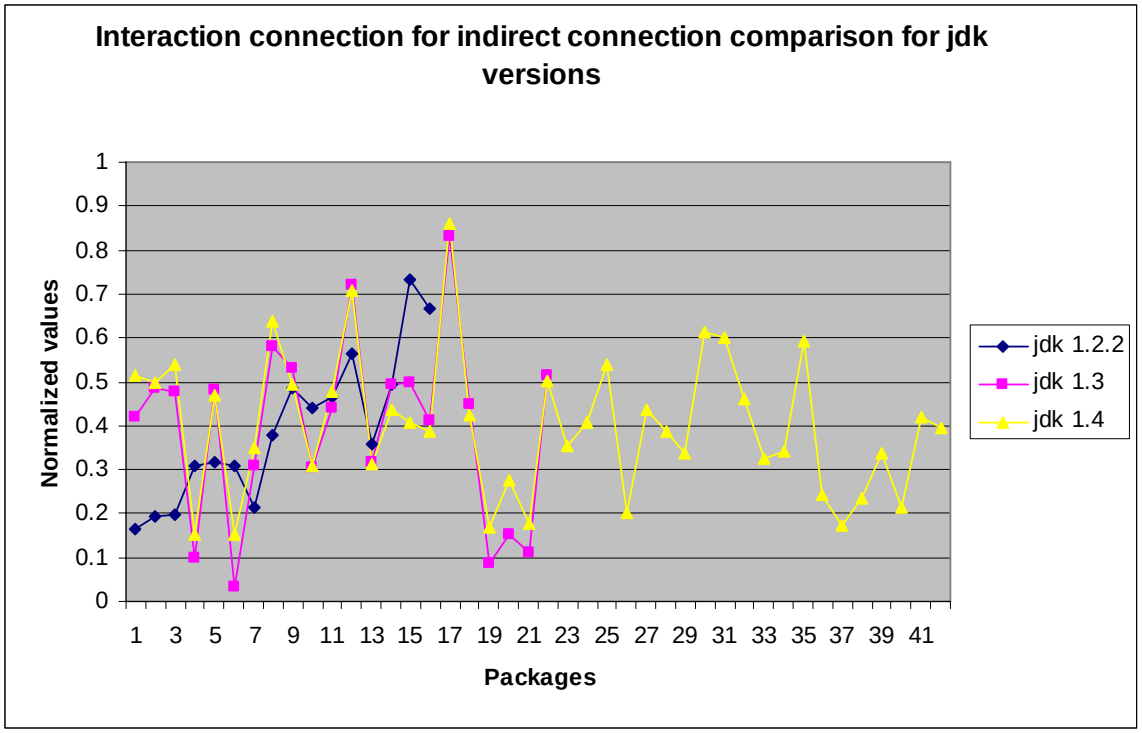

Figure 5.29: Comparison Graph for JDK version involving Indirect Connections

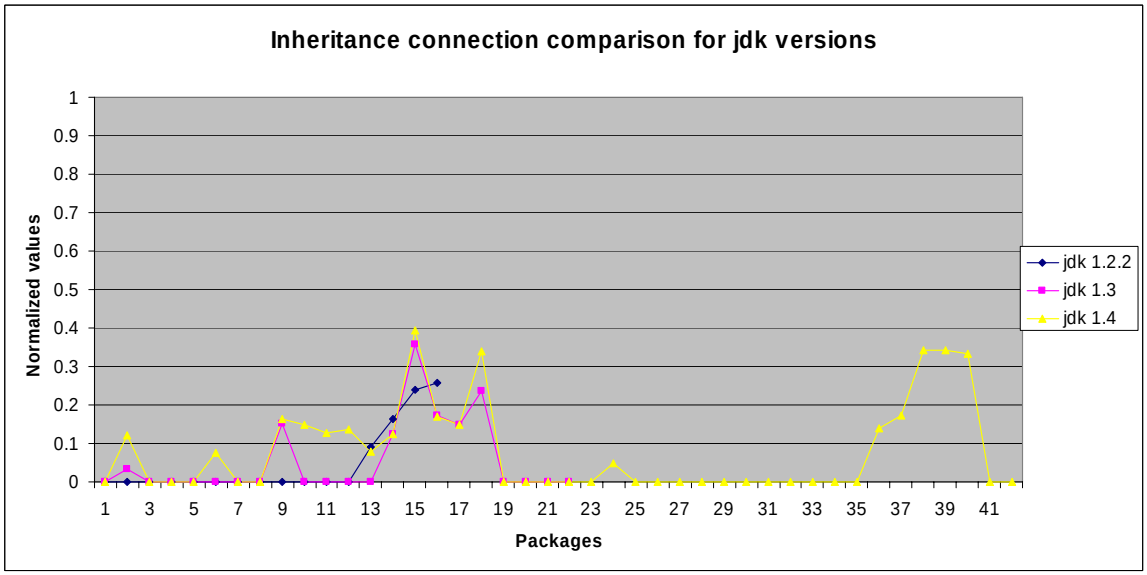

Figure 5.30: Package Inheritance Cohesion Graph for JDK versions

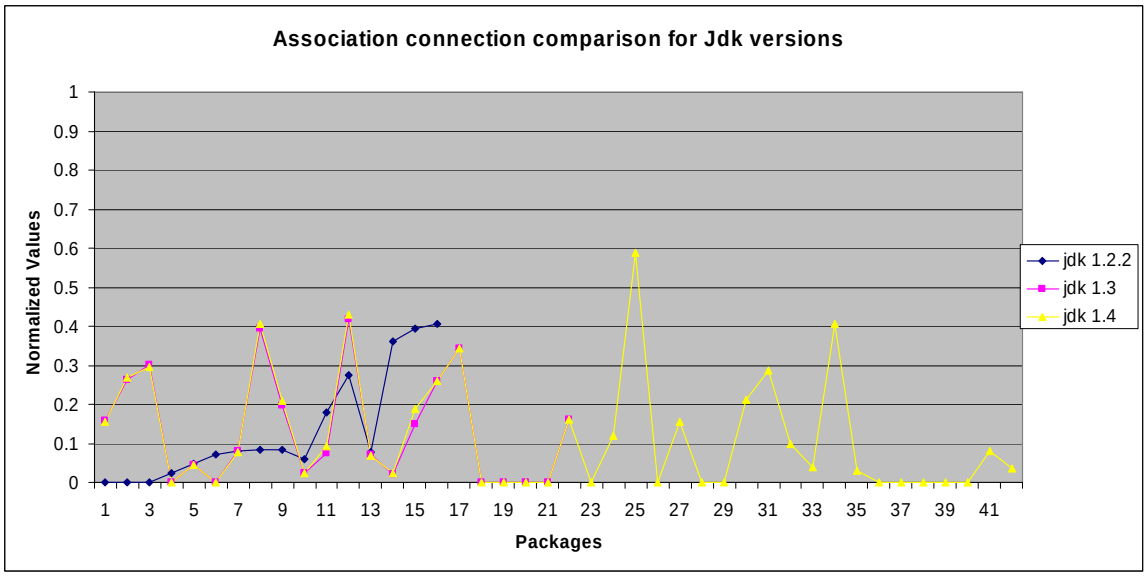

Figure 5.31: Package Association Cohesion for JDK versions

Package javax.swing.border Description: Provides classes and interface for drawing specialized borders around a Swing component. You can subclass these classes to create customized borders for your components instead of using the default borders provided by the look-and-feel being used.

Package javax.swing.undo Description: Provides support for undo/redo capabilities in an application such as a text editor. It is for developers that provide undo/redo capabilities in their application.

Package javax.swing.tree Description: Provides classes and interfaces for dealing with java.awt.swing.JTree. You use these classes and interfaces if you want control over how trees are constructed, updated, and rendered, as well as how data associated with the tree nodes are viewed and managed.

After the brief Introduction of the packages which are displaying lower cohesion values we now look at the changes that Jdk 1.3 has undergone with respect to 1.2.2. Javax. Swing is one of the packages which have undergone a massive change since its inception

in Jdk1.2.2, for example, the class ‘border’ in javax.swing.border, and the class AbstractLayoutCache in javax.swing.tree and, the class UndoManager in javax.swing.undo package, the Serialized objects of these classes are not compatible with subsequent Swing releases. The serialization support is appropriate for short term storage or RMI between applications running the same version of Swing. Also the number of bugs in the Jdk 1.3 version for the javax.swing package is 15 as per the official records on Sun website which is the highest reported for that particular version. The following is the performance analysis of javax.swing package for Jdk 1.2.2 found on Sun‘s Website and the corrections that have been done in order to fix the problem. “One of the main reasons that Swing's startup performance was slower than desired was that as soon as any component requires a UI delegate, the UIManager loads a look and feel, which results in loading a defaults table which includes defaults for UIs for all component classes. In previous releases, we mistakenly believed that instance creation should be avoided, and so we delayed instance creation by creating anonymous implementations of LazyValue, an interface which acts as a lightweight proxy that only creates its instance the first time it is retrieved from the defaults table. Performance analysis for Kestrel indicates that we were wrong in believing that instance creation was the determining factor. In fact, the overwhelming factor contributing to delay and increased footprint in this area was classloading, which was ironically not helped by our creation of lots of anonymous interface implementations. The general approach taken to fix this was to define a concrete LazyValue implementation in UIDefaults.java which uses reflection to create its proxied instance when asked to do so. This class is called

UIDefaultProxy. As a result only one class is loaded, and about 90 other classloads could be avoided in a Hello World example".

Therefore one of the reasons that can be attributed to the high package cohesion values in Jdk 1.2.2 is actually due to the creation of lots of anonymous interface implementations. Apart from the interaction cohesion values the metric results for these three packages are improved when looked at the graph depicting the addition of indirect connections and also for the inheritance and association cohesion, the values seems to be improving in the later versions of Jdk. Sami And Ville [39] have conducted a case study using the LCOM metric on the packages java.lang, java.util, java.io, java.awt, java.net, and javax.swing and their sub packages. They also ran the metric for the JDK demo files and for the whole JDK .Trivial classes are classes that do not have any methods or instance variables, the metric is meaningless for them so they are ignored. The results showed that a large part of the classes were trivial. Also, many of the classes have the best possible cohesion value (0.0). These are mostly small classes. Of the other classes, most have cohesion value over 0.5. Many classes have values that are close to 1.0, the worst value. Note here that LCOM[19][20] is an inverse metric so the higher the cohesion value the lower will be its cohesion and vice versa and it measures cohesion of class. The following table shows the cohesion values using LCOM for javax.swing package in [39].

Range	NOC
Trivial	672
0.0	150
0.0 - 0.1	3
0.1 - 0.2	6
0.2 - 0.3	12
0.3 - 0.4	11
0.4 - 0.5	10
0.5 - 0.6	51
0.6 - 0.7	48
0.7 - 0.8	54
0.8 - 0.9	117
0.9 - 1.0	165
1.0	89

Table 5.1: Cohesion Values Using Lcom for Javax.Swing Package

Here NOC means the number of classes that are on that range. The ranges are closed at the lower end and open at the higher end point. For example the peak at the range 0.5-0.6 is mostly caused by the classes that have value 0.5. The possible reasons outlined for such behavior is divided into three categories. First, the class has low cohesion because the definition of LCOM does not measure cohesion correctly, second, because the class has been designed or implemented badly, and third, the class has low cohesion because its design requires so.

5.5 Implementation of Giancarlo metric using our connection types

In this section we discuss the results of JDK versions for the different connection categories specified in chapter 4 but using the Giancarlo metric. The details of the Giancarlo metric can be found in chapter 2. A quick refresh of the Giancarlo metric is as follows:

The metric is defined as the number of internal classes to which a class is coupled normalized with the number of the possible coupling relationship among the classes: *m(m-1)*.

$$CC = \sum_{i=1}^{m} \sum_{j=1, j\#i}^{m} h(Ci, Cj) / m(m-1)$$

Where h (C_i, C_j) = 1 if C_i and C_j are coupled

0 otherwise

Figures 5.32 & 5.33 show the comparison between Giancarlo & Our metric for JDK 1.2.2 for Interaction connection both for direct and Indirect Connection types. It can be noticed here that the Interaction cohesion values for JDK 1.2.2 using Giancarlo metric are almost same but it can also be seen that are slightly higher when compared to PIC metric in most of the cases. Similar trend is true for other versions of JDK for all connection categories. However for indirect connection type, as expected the indirect cohesion values are greater than direct connection values. But a point to notice here is that the indirect connection values for PIC metric is greater than indirect connection cohesion values using Giancarlo metric. The same trend can be seen for Indirect Connection values for all version of JDK. The following graphs reflect these observations.

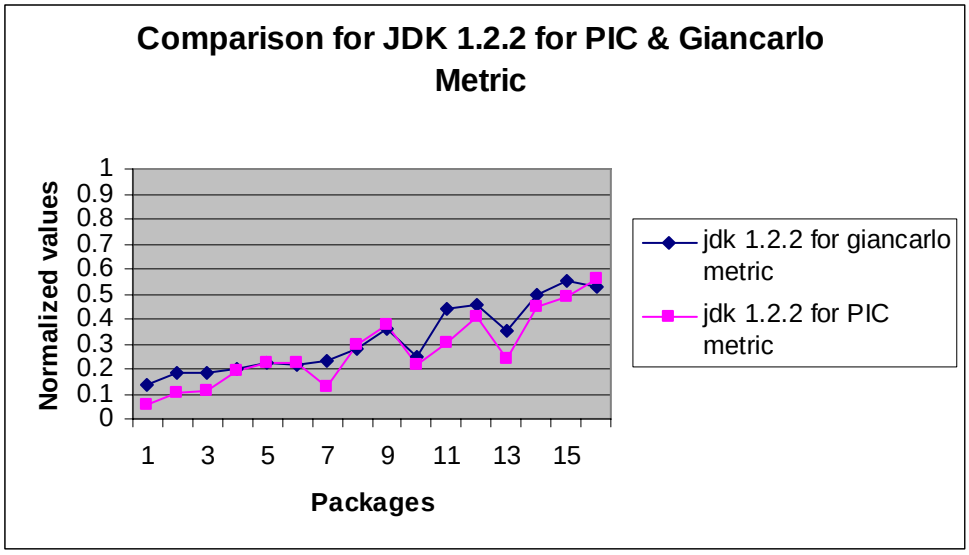

Figure 5.32: Comparison graph for JDK 1.2.2 for Package interaction cohesion with Giancarlo metric

Figure 5.33 depicts the comparison between PIC & Giancarlo metric for JDK 1.2.2 for indirect connection types. The Indirect connections values are greater when compared to direct connection types as expected however when compared to PIC metric the values for Giancarlo metric are lower.

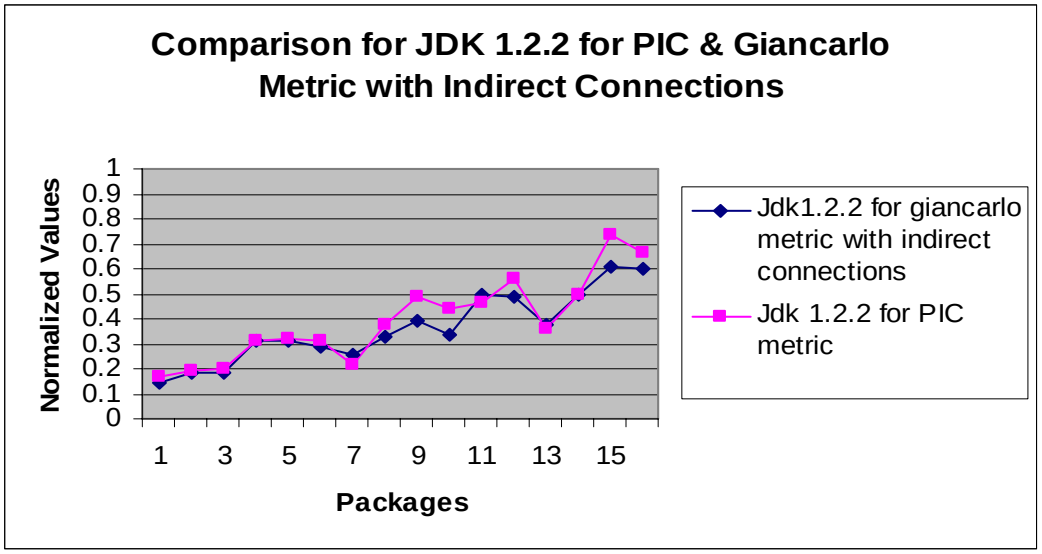

Figure 5.33: Comparison graph for JDK 1.2.2 for Package interaction cohesion involving indirect connection with Giancarlo metric

Figure 5.34 shows the Association cohesion values comparison between Giancarlo & PAC metric. As can be seen the Association cohesion values for Giancarlo metric are slightly greater than PAC metric in some cases. This trend is almost reflected in all of the connection categories for all versions of JDK except for indirect connection types.

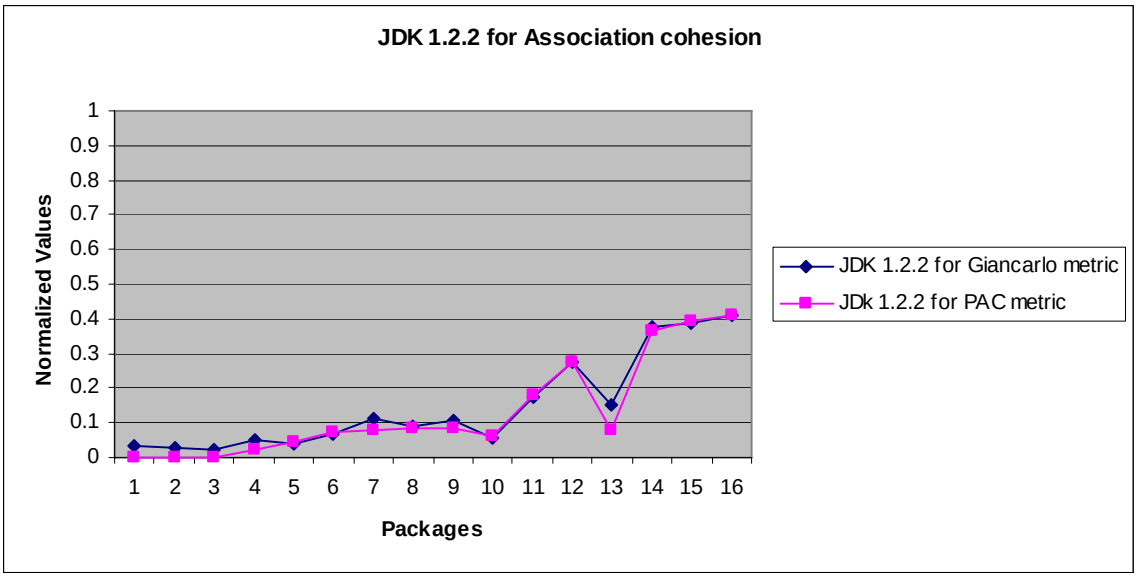

Figure 5.34: Comparison graph for JDK 1.2.2 for Package Association cohesion with Giancarlo metric

Figure 5.35 shows the Inheritance cohesion values comparison between Giancarlo & PInC metric.

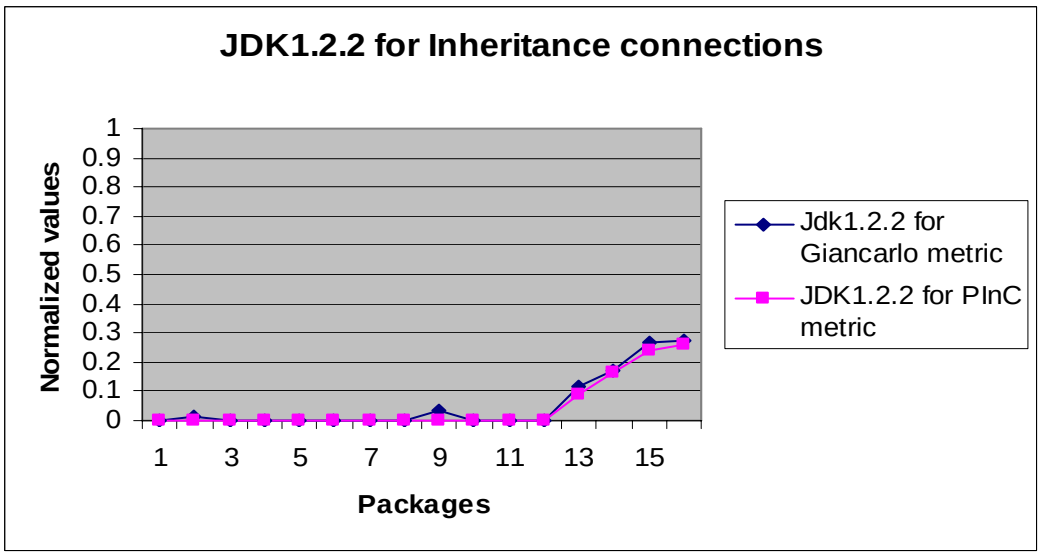

Figure 5.35: Comparison graph for JDK 1.2.2 for Package inheritance cohesion with Giancarlo metric

Figures 5.36 & 5.37 show the comparison between Giancarlo & Our metric for JDK 1.3 for Interaction connection for both Direct and Indirect Connection types.

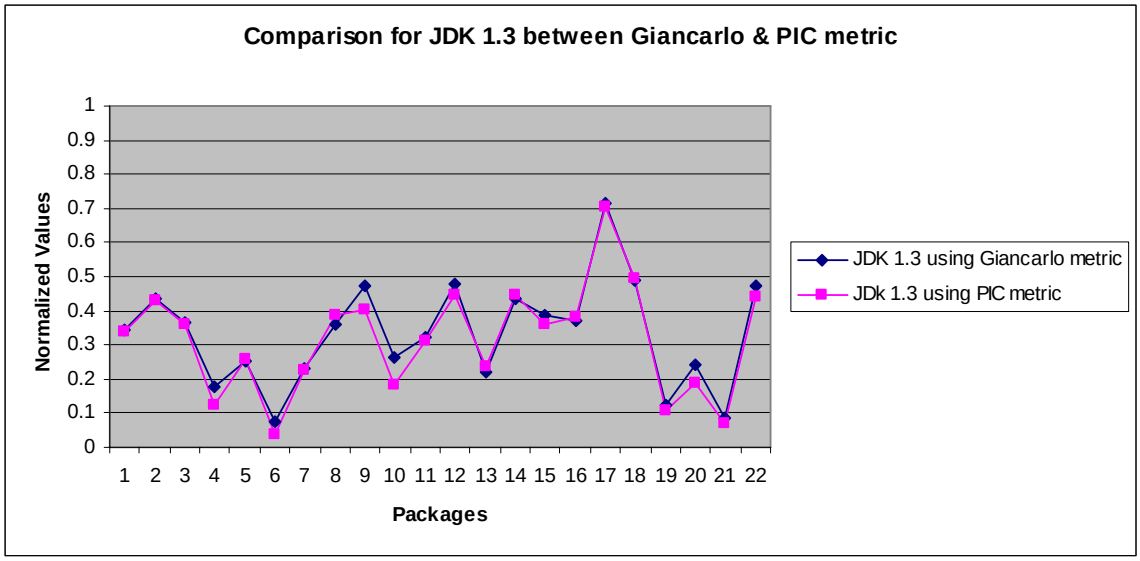

Figure 5.36: Comparison graph for JDK 1.3 for Package interaction cohesion with Giancarlo metric

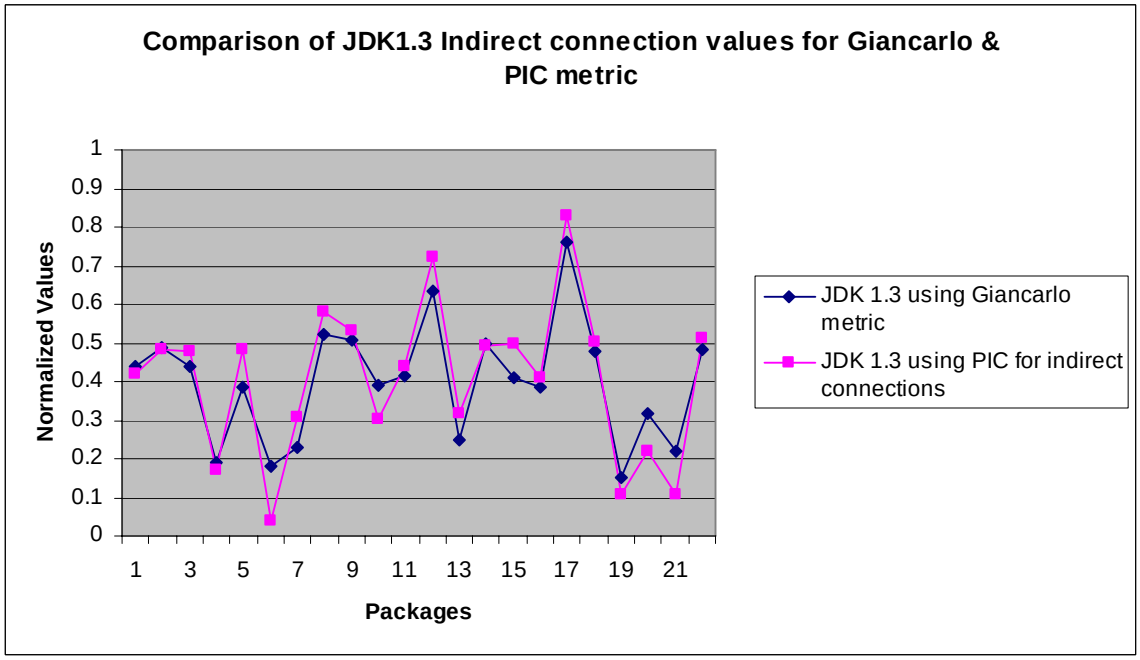

Figure 5.37: Comparison graph for JDK 1.3 for Package interaction cohesion involving indirect connections with Giancarlo metric

Figure 5.38 shows the Inheritance cohesion values comparison between Giancarlo & PInC metric

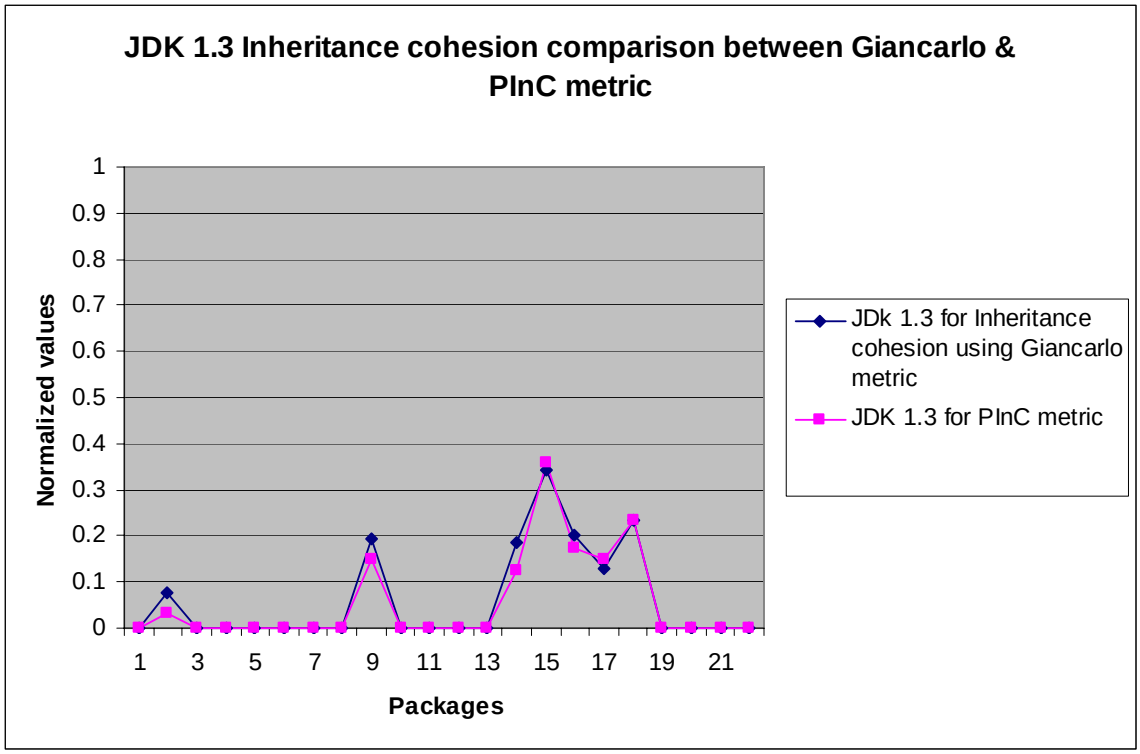

Figure 5.38: Comparison graph for JDK 1.3 for Package interaction cohesion with Giancarlo metric

Figure 5.39 shows the Association cohesion values comparison between Giancarlo & PAC metric for JDK 1.3

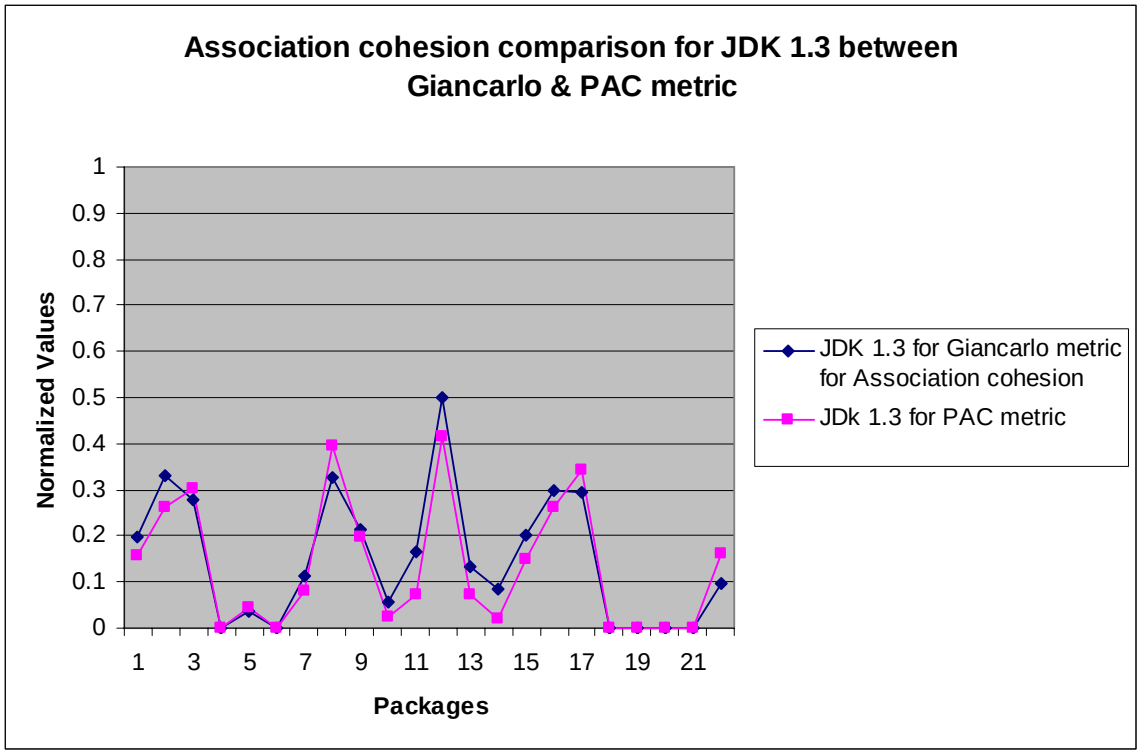

Figure 5.39: Comparison graph for JDK 1.3 for Package Association cohesion with Giancarlo metric

Figures 5.40 & 5.41 shows the comparison between Giancarlo & Our metric for JDK 1.4 for Interaction connection.

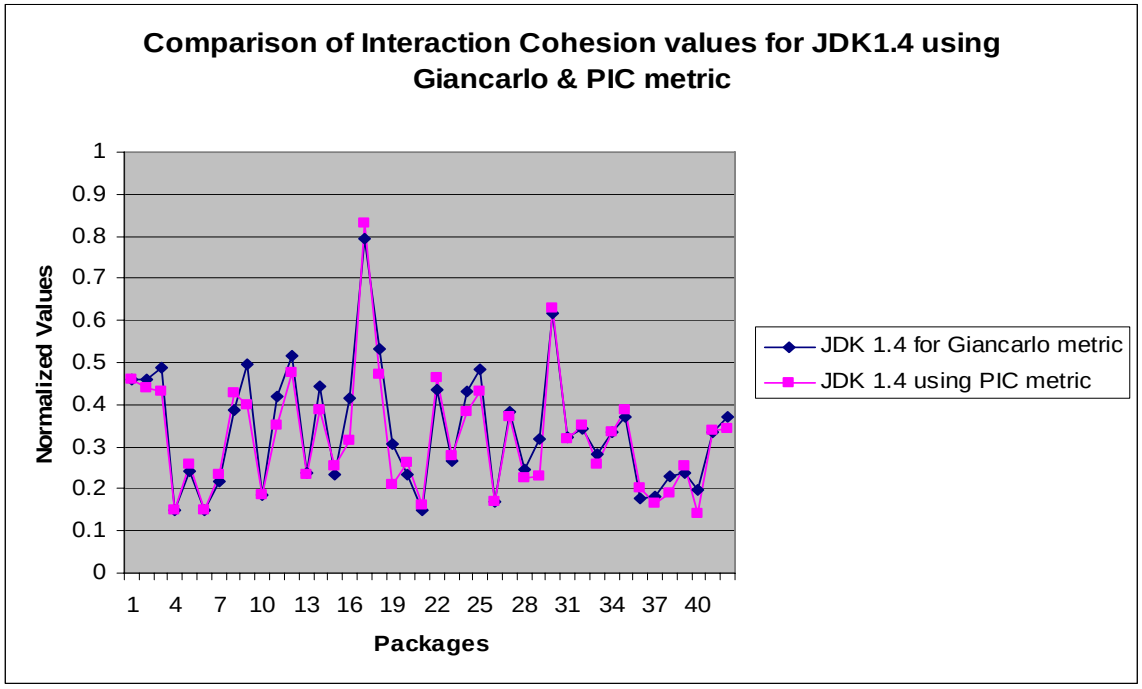

Figure 5.40: Comparison graph for JDK 1.4 for Package interaction cohesion with Giancarlo metric

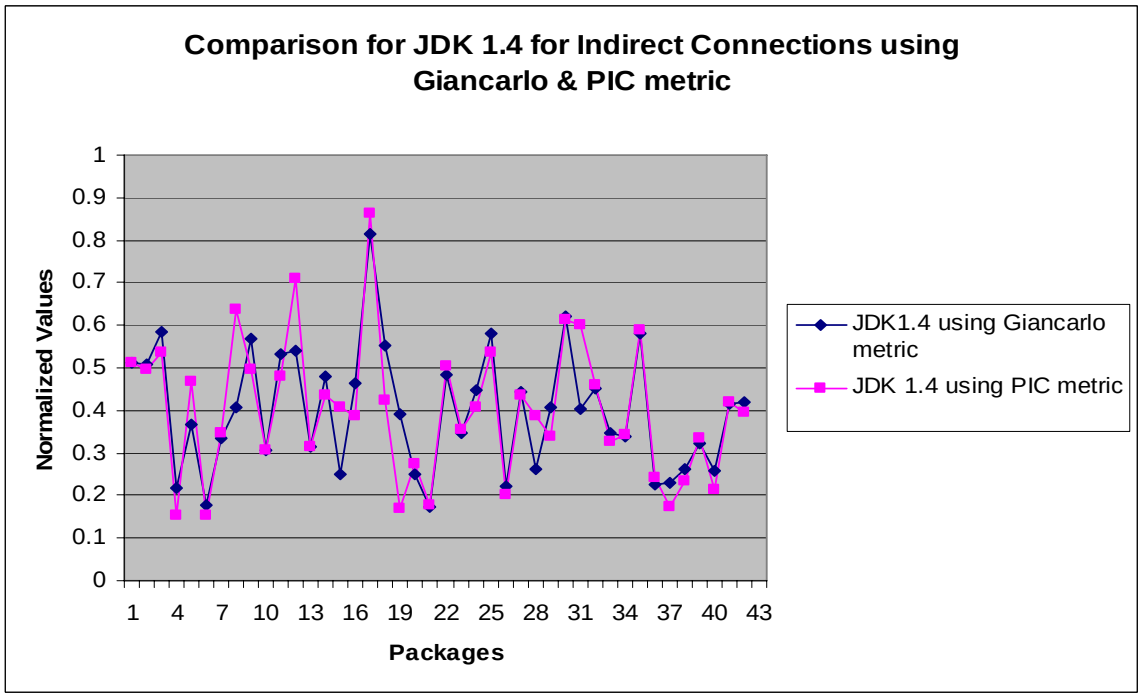

Figure 5.41: Comparison graph for JDK 1.4 for Package interaction cohesion involving indirect connection with Giancarlo metric

Figure 5.42 shows the Inheritance cohesion values comparison between Giancarlo & PInC metric for JDK 1.4.

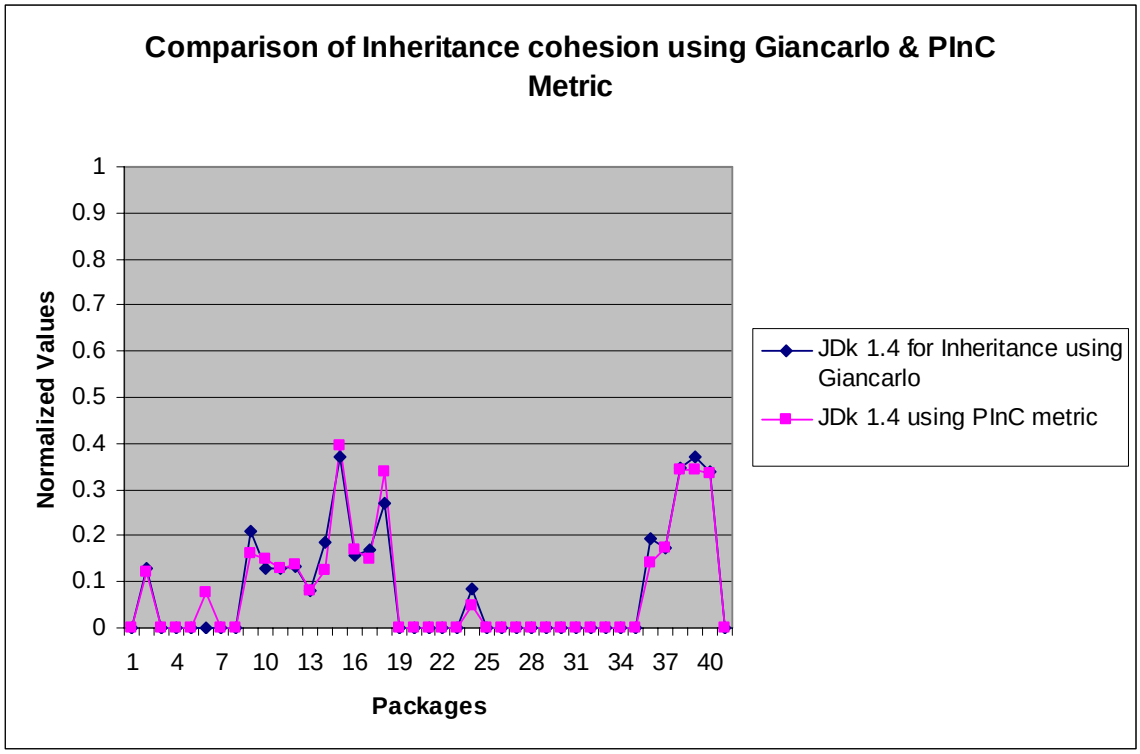

Figure 5.42: Comparison graph for JDK 1.4 for Package inheritance cohesion with Giancarlo metric

Figure 5.43 shows the Association cohesion values comparison between Giancarlo & PAC metric for JDK 1.4. It is shown below.

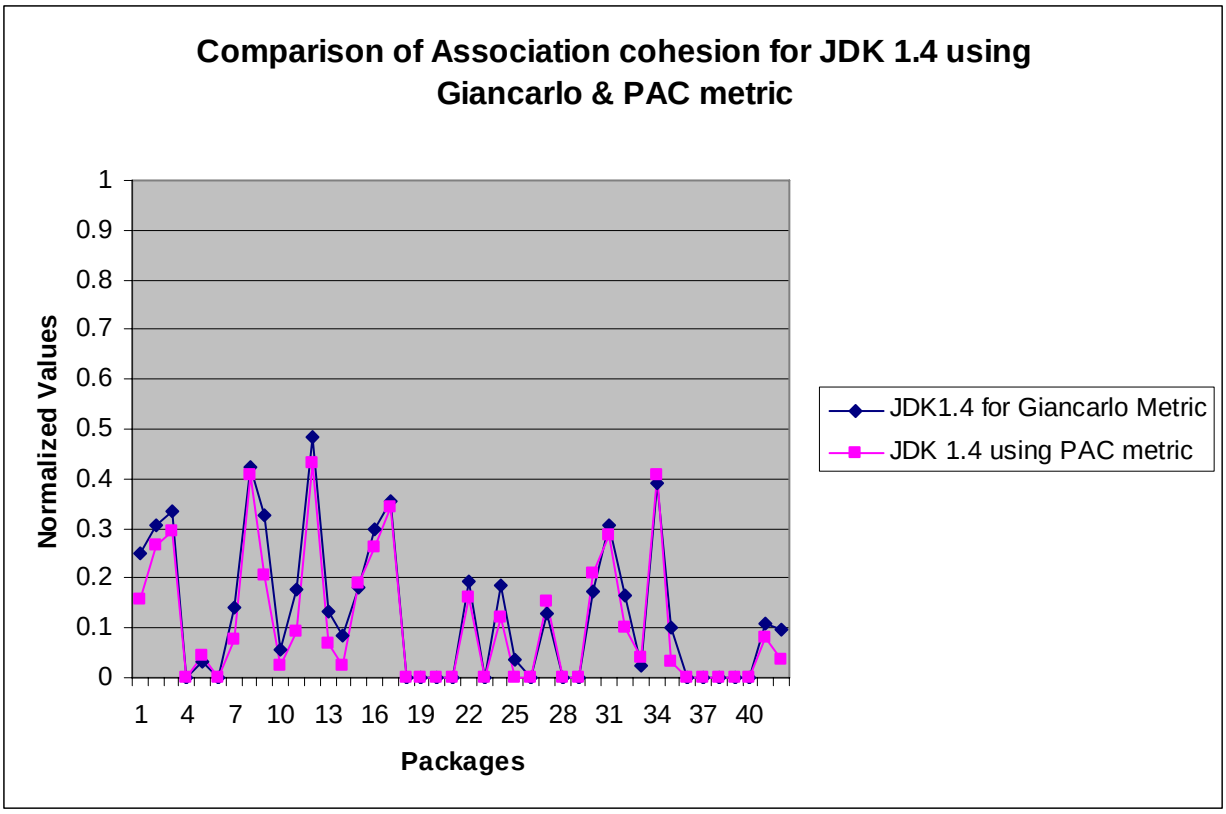

Figure 5.43: Comparison graph for JDK 14 for Package Association cohesion with Giancarlo metric

5.6 Summary of Comparison Results:

The Comparison results for the new package cohesion metric and the Giancarlo metric using our connection types show that the package cohesion metric values for interaction , inheritance, and association cohesion are all slightly greater in most of the packages for all versions of JDK for Giancarlo metric as against to our metric. However, the indirect connection values for Giancarlo metric are less than that for our metric and this behavior is consistent for all versions of JDK. The reason for such behavior is due to the fact that we are considering nodes as connection criteria between two classes; see chapter 4, and section 4.4.1 example , whereas in Giancarlo metric we count an edge between two

methods of two different classes as a connection. To visualize the situation we give an example as follows:

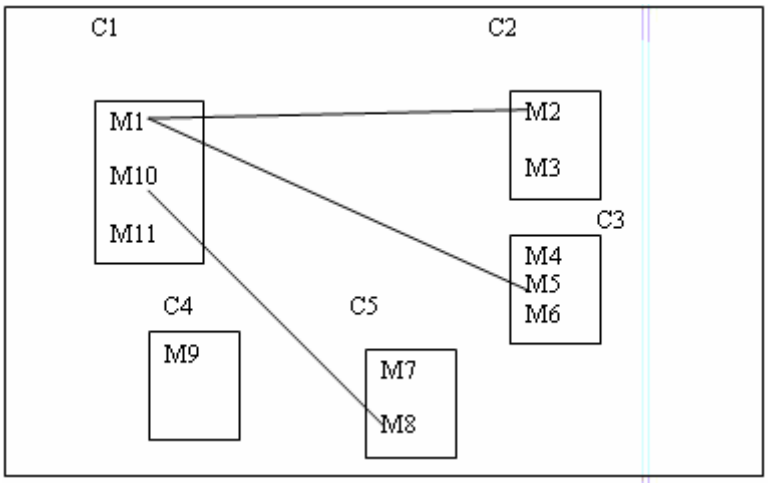

Figure 5.44: Example figure 1 to illustrate results

In this figure , the methods involved in interaction for class c1 are two, in which case the package cohesion value becomes,

$$\frac{2 \text{ (No. of methods having connection)} +1 +1 + 1 +0}{3 \text{ (Total no. of methods in the class)} + 2 +3 +2 +1} = 0.45$$

Whereas in figure below (figure 5.44) we have removed a connection from method M10 and instead added the same connection from method M1, in which case the package cohesion value reduces, which is not the case with Giancarlo metric as it considers the edges as connection.

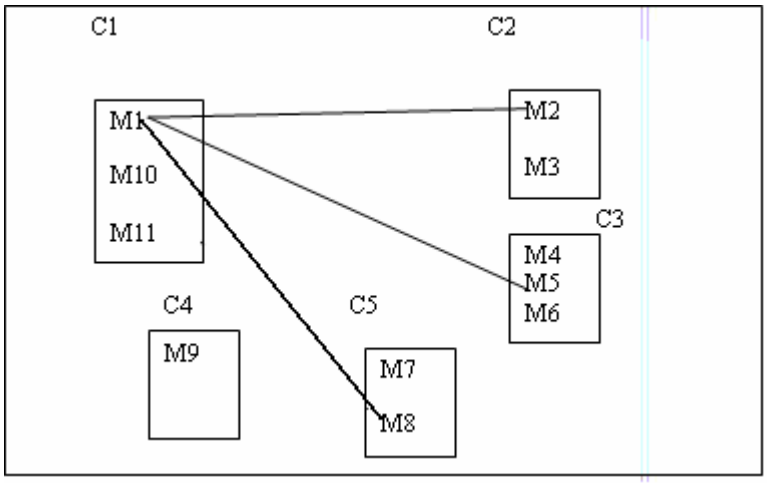

Figure 5.45: Example figure 2 to illustrate results

$$\frac{1(\text{No. of methods having connection}) + 1 + 1 + 1 + 0}{3\,(\text{Total no. of methods in the class}) + 2 + 3 + 2 + 1} = 0.36$$

Therefore, it is clear from this example that the cohesion values for our metric depends on how many methods in a class have a connection with other classes, the more the number of methods involved in interaction the higher the value of the metric will be and vice versa.

Whereas for the indirect connections as we count the nodes as opposed to edges in Giancarlo metric so an indirect connection between methods will result in adding only one connection in Giancarlo metric as opposed to adding 2 connections in our metric as we count the nodes (both the methods involves in indirect connection), which explains the slightly greater values that we are getting for Indirect connection values for our metric as opposed to the Giancarlo metric.

5.7 Analyzing JDK versions using statistics

PIC	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.27	0.31	0.32
maximum	0.55	0.704	0.83
Top 10%	0.48	**0.44**	0.48
Top25%	0.4	0.42	0.46
Bottom25%	0.13	0.19	0.22
Bottom 10%	0.1	0.1	0.16
minimum	0.05	0.03	0.142

Table 5.2: Statistical analysis for JDK versions for PIC metric

PIC (Indirect)	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.369	0.397	0.401
Maximum	0.73	0.83	0.86
Top 10%	0.66	**0.57**	0.60
Top25%	0.492	0.498	0.496
Bottom25%	0.21	0.3	0.30
Bottom 10%	0.08	0.09	0.17
Minimum	0.04	0.03	0.15

Table 5.3: Statistical analysis for JDK versions for PIC metric with Indirect Connections

PInC	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.04	0.05	0.07
maximum	0.25	0.235	0.395
Top 10%	0.23	**0.17**	0.33
Top25%	0.08	0.125	0.14
Bottom25%	0	0	0
Bottom 10%	0	0	0
minimum	0	0	0

Table 5.4: Statistical analysis for JDK versions for PInC metric

PAC	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.13	0.134	0.18
Maximum	0.407	0.417	0.58
Top 10%	0.39	**0.34**	0.38
Top25%	0.27	**0.26**	0.29
Bottom25%	0	0	0
Bottom 10%	0	0	0
minimum	0	0	0

Table 5.5: Statistical analysis for JDK versions for PAC metric

PIC	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.31	0.314	0.337
Maximum	0.54	0.61	0.79
Top 10%	0.52	**0.47**	0.538
Top25%	0.425	**0.41**	0.43
Bottom25%	0.187	0.22	0.21
Bottom 10%	0.181	0.163	0.18
Minimum	0.103	0.07	0.125

Table 5.6: Statistical analysis for JDK versions for PIC metric using Giancarlo metric

PIC (Indirect)	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.36	0.38	0.39
maximum	0.61	0.75	0.81
Top 10%	0.602	**0.52**	0.56
Top25%	0.5	**0.46**	0.52
Bottom25%	0.27	0.23	0.26
Bottom 10%	0.18	0.19	0.23
minimum	0.1	0.15	0.15

Table 5.7: Statistical analysis for JDK versions for PIC metric with Indirect Connections using Giancarlo metric

PInC	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.06	0.06	0.07
maximum	0.31	0.34	0.37
Top 10%	0.27	**0.2**	0.21
Top25%	0.08	0.12	0.13
Bottom25%	0	0	0
Bottom 10%	0	0	0
minimum	0	0	0

Table 5.8: Statistical analysis for JDK versions for PInC metric using Giancarlo Metric

PAC	JDK 1.2.2	JDK1.3	JDK1.4
Average	0.15	0.15	0.12
maximum	0.46	0.45	0.48
Top 10%	0.36	**0.32**	0.384
Top25%	0.27	0.27	0.298
Bottom25%	0.03	0	0
Bottom 10%	0.022	0	0
minimum	0.02	0	0

Table 5.9: Statistical analysis for JDK versions for PAC metric using Giancarlo Metric

The tables 5.2, 5.3, 5.4, 5.5 illustrates the Average, Top 25% i.e., the top quartile, Top 10%, Bottom 25% or the bottom quartile and the maximum and minimum values for different versions of JDK, namely JDK 1.2.2, JDK 1.3 & JDK 1.4 for the three metrics Package Interaction cohesion (PIC), PIC with indirect connections Package Inheritance cohesion (PInC) and the Package Association cohesion (PAC).

The mean and standard deviation are useful to summarize a set of observations. When the data have a skewed distribution it is often preferable to quote instead the median. The first and third quartiles (25th and 75th centiles) are sometimes used; these define the interquartile range. This statistical analysis gives a more detailed

understanding of the behavior of the packages and helps in corroborating the results to the metrics developed. The Data is arranged in ascending order of the cohesion values of packages. As can be seen the Top quartile values for PIC is 0.4 for JDK 1.2.2 which means that top25% of the packages have value of 0.4 and this value is better in the later versions of JDK, for example JDK 1.4 has top 25% of the packages with value of 0.46. The tables 5.6 to 5.9 illustrate the statistics for the Giancarlo metric using our connection criteria. The metric also depicts a similar behavior i.e., the average is also increasing for the later versions when compared to their previous counterparts, same is true for Bottom quartile , maximum & minimum for all the categories for all the versions. This confirms that with later releases the cohesion in the packages for JDK has been improved, and therefore corroborates the findings of the PIC, PInC & PAC metrics discussed in the previous sections, that with the newer versions the cohesion values of the packages is showing an upward trend in most of the packages. However an important point to note here for both the metrics is that the Top 10 % values of the packages for JDK 1.3 are found to be lower when compared to JDK version 1.2.2. This can be attributed to the fact that JDK 1.3 has undergone a major reconstruction both in terms of the number of classes added when compared to JDK 1.2.2 and also it has undergone major reconstruction [43]. Grosser et.al, mentioned in their research that the no. of classes in JDK 1.2 version were 580 and in JDK 1.3 the number of classes were 2158, the jump in the number of classes in the version 1.3 with respect to 1.2 is the result of the integration into the standard API of two previously independent libraries, Swing and CORBA [43]. Thus, it can be asserted that the cohesion values will be affected when there is a major reconstruction process in the system. Thus, the usefulness of the metric also comes from the fact that it shows

whether addition of classes increases the cohesion of a package or not, as was observed in most of the cases that the later versions of Jdk showed improvement with respect to the cohesion values, but this is not always the case i.e., for some packages the cohesion values decreased which are illustrated in section 5.4.4 with the possible reasons. So in this case It should ring an alarm to the designer that whether the changes to the system, either addition or removal of classes has led to decrease in cohesion values and he can then introspect into the system. Another observation that is consistent with all the systems tested is that the Inheritance and Association-Aggregation values are very less , and it should be seen as to whether they were intended to be used that way to their advantage or not. Also, indirect connections are of potential interest when defining criteria for when to break up a class. In class c, each method is directly or indirectly connected with every other method (the vertices are the methods). In Figure 5.45, class d, on the other hand, there are pairs of methods which are not even indirectly connected. This may indicate that the methods should not be encapsulated in the same class. Note, however, that there could be other reasons why the methods should be encapsulated together in one class anyway, e.g., because of method invocations from one connected component to the other. It is evident from the Indirect Connection values, for all the versions of JDK that similar classes have been encapsulated in the right packages as they do not indicate the behavior depicted in figure 5.45.

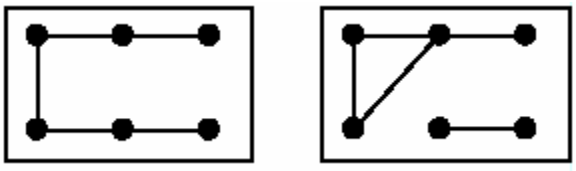

Figure 5.46: Indirect Connections Example

5.7.1 Analyzing the previous systems using Quartiles

PIC	Jext	Saxon6.5.2	Saxon 8	Babeldoc 1.0	JDK1.4(our Metric)
Average	0.25	0.29	0.33	0.28	0.32
Maximum	0.87	0.506	0.56	0.7	0.83
Top 10%	0.46	0.4	0.52	0.52	0.48
Top25%	0.35	0.35	0.42	0.42	0.46
Bottom25%	0.125	0.2	0.25	0.15	0.22
Bottom 10%	0.1	0.14	0.17	0.08	0.16
Minimum	0.04	0.09	0.12	0.04	0.142

Table 5.10: Statistical analysis for PIC values for Jext, Saxon and Babeldoc, JDK1.4

PInC	Jext	Saxon6.5.2	Saxon 8	Babeldoc 1.0	JDK1.4
Average	0.04	0.044	0.07	0.031	0.07
Maximum	0.33	0.25	0.36	0.14	0.395
Top 10%	0.24	0.1	0.28	0.08	0.33
Top25%	0.06	0.06	0.13	0.02	0.14
Bottom25%	0	0	0	0	0
Bottom 10%	0	0	0	0	0
Minimum	0	0	0	0	0

Table 5.11: Statistical analysis for PInC values for Jext, Saxon and Babeldoc, JDK1.4

PAC	Jext	Saxon6.5.2	Saxon 8	Babeldoc 1.0	JDK1.4
Average	0.049	0.11	0.13	0.04	0.18
Maximum	0.54	0.42	0.437	0.27	0.58
Top 10%	0.11	0.36	0.38	0.18	0.38
Top25%	0.05	0.24	0.22	0.08	0.29
Bottom25%	0	0	0	0	0
Bottom 10%	0	0	0	0	0
Minimum	0	0	0	0	0

Table 5.12: Statistical analysis for PIC values for Jext, Saxon and Babeldoc, JDK1.4

The tables 5.10, 5.11, 5.12 illustrates the Average, Top 25% i.e., the top quartile, Top 10%, Bottom 25% or the bottom quartile and the maximum and minimum values for Jext, Saxon 6.5.2, Saxon 8.0, and Babeldoc 1.0 & JDK 1.4 for PIC , PInC and PAC metrics. The Saxon versions also confirms the findings that the values for PIC, PInC and PAC

values are increasing from the previous to later versions which was as expected and similar trend depicted by JDK versions too. As explained in the section above (section 5.6) the cohesion values for our metric depends on how many methods in a class have a connection with other classes, the more the number of methods involved in interaction the higher the value of the metric will be and vice versa. Both the metrics depict almost similar values for all the JDK version tested against ,with Giancarlo metric having slightly greater values, this shows that functionally the metrics captures the same information. Both the metrics are theoretically validated.

Chapter 6

Conclusions and Future Work

In this chapter we present a summary of the contribution of this thesis, outline the limitation of the work and provide suggestion on how it can be improved in the future.

6.1 Summary and Contributions of the Thesis

When designing systems, we strive to achieve a high degree of cohesion, designing elements that focus on performing a single task. But the research on cohesion has only been limited at the class level, whereas the cohesion at the package level is also of utmost importance. As is clearly evident from the literature survey conducted in chapter 2 only one research paper was found on package cohesion which shows that very little focus has been given to package cohesion measurement which in turn helps in analyzing the maintainability and reusability of software and also can save lots of money in future course of software advancement or restructure. We first researched the connection types that are of importance in affecting the cohesiveness of a package and after an exhaustive search we came up with a list of 12 connection types, see chapter 4, that are of importance in determining the package cohesion.

We then came up with a metric which helps in capturing the cohesiveness of a package using these connection types and gives an idea of the packages which needs attention with respect to the low cohesion values. The metrics are proposed in chapter 4 which has been investigated thoroughly in order to capture the connections which are of interest to capture the cohesiveness of a package. The various factors considered are the interaction connections including the indirect connections along with inheritance

relationships represented as a set of 4 connection tables in our connection table in chapter 4 and also Association and Aggregation relationship again represented as a set of 3 connection types which shows that the relationships are also connections after all. The table has been thoroughly investigated in order not to miss out any connection that relates to package cohesion keeping in mind not to have any redundant connection type. Thus we have a total of three metrics emphasizing the three connection criteria discussed, the package Interaction metric (PIC), the package Inheritance metric (PInC) and the package Association metric (PAC).

To test the accuracy of our metric we ran experiments on varied projects starting with not so popular ones to the most renowned and widely used Jdk software system in order to test the validity of the metric in accordance to the standards set by Jdk suite in the software development with respect to its widely accepted design and implementation. The metric was tested with respect to four software systems including the different versions of Jdk so as to see the variation of the cohesiveness of the packages over different versions. The results are explained in chapter 5. We also implemented the Giancarlo metric using our connection criteria listed in chapter 4 to compare the results of our metric with Giancarlo metric. The metric used using the connection criteria investigated showed similar results to our metric and in some cases the values were slightly greater than our metric, see chapter 5 for details. The metric is theoretically validated and the results for the case studies for JDK versions are also promising. So to summarize, the Giancarlo metric with the exhaustive list of our connection criteria for Package cohesion for the metric serves the purpose of calculating the cohesion of a package. However an important point to note here is that the Top 10 % values of the

packages for JDK 1.3 are found to be lower when compared to JDK version 1.2.2. This can be attributed to the fact that JDK 1.3 has undergone a major reconstruction both in terms of the number of classes added when compared to JDK 1.2.2 and also it has undergone major reconstruction for Javax.Swing package [43]. Since, the metric under scrutiny calculates the cohesion as the number of connections over the number of methods in the classes, and since the no of classes added has been enormous, nearly 400%, the denominator value has increased significantly and the lower cohesion value for Top 10% indicates that the numerator, which calculates the number of connections has not been increased proportionately. Grosser et.al, mentioned in their research that the no. of classes in JDK 1.2 version were 580 and in JDK 1.3 the number of classes were 2158, the jump in the number of classes in the version 1.3 with respect to 1.2 is the result of the integration into the standard API of two previously independent libraries, Swing and CORBA [43]. Thus, it can be asserted that the cohesion values will be affected when there is a major reconstruction process in the system. The usefulness of the metric comes from the fact that it shows whether addition of classes increases the cohesion of a package or not, i.e., if the cohesion value is less as was shown for some packages of JDK, see section 5.4.4 then the designer should introspect the system to look whether the changes made, either addition or removal of classes has led to decrease in cohesion values, this in effect helps to verify the cohesiveness of the package. Another observation that is consistent with all the systems tested is that the Inheritance and Association-Aggregation values are very less , and it should be seen as to whether they were intended to be used that way to their advantage or not. The following points summarize the findings.

1. The experimentation for JDK has shown that the cohesion has decreased when there is a major reconstruction process in the system.

2. The results confirms the findings that the values for PIC, PInC and PAC values for by JDK versions are increasing from the previous to later versions which was as expected and similar trend depicted by other systems too.

3. However, another observation that is consistent with all the systems tested is that the Inheritance and Association-Aggregation values are very low, even thought the later versions are showing slight improvements when compared to the previous versions, as mentioned above.

4. It is evident from the Indirect Connection values, for all the versions of JDK that similar classes have been encapsulated in the right packages.

5. The Giancarlo metric with the exhaustive list of our connection criteria for Package cohesion for the metric serves the purpose of calculating the cohesion of a package.

6.2 Limitations & Future Work

Though we attempted to run experiments on variety of systems, see chapter 5 still we cannot say that the metric is empirically validated as we need information about the systems undergoing tests and the reliability & dependability of those systems. Like for example there is no such concrete information in the literature about the cohesiveness of the systems tested like Saxon or Babeldoc such that it can be said that the results of our metric collaborate to such findings, similar is the case for JDK versions even though they are believed by many researchers and developers to have very good design but still there cannot be any 100% assurances, so the most important thing is to have the data in order to carry out the required tests and the analysis of data is the foremost step in dealing with

the empirical validation of any measure [40]. The Future work would be to devise a metric that measures the architectural stability as the package cohesion will play a vital part in assessing the stability of one's architecture.

7 References

[1] Fenton E. N., Pfleeger S. L., *Software Metrics – A Rigorous & Practical Approach,* PWS Publishing company, Boston, 1997.

[2] Chae H. S., Kwon Y. R., Bae D., "A Cohesion Measure for Object-Oriented classes", *Software-Practice & Experience*, v.30, n.12, pp.1405-1431, Oct. 2000.

[3] Larman C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Desing and the Unified Process*, Prentice-Hall, Inc. US, 2002.

[4] Eder J., Kappel G., Schrefl M., "*Coupling and Cohesion in Object-Oriented Systems*", Technical Report of University Klagenfurt, Institute of Computer Science, 1993.

[5] Kevlin Henney, original English draft of Patterns in Java column *published as Kapselung in JavaSPEKTRUM*, July–August 2003.

[6] Jon Hopkins, "Component Primer", *Communications of the ACM*, vol.43, no.10, October 2003.

[7] OMG (Object Management Group) Specification, UML (Unified Modeling Language): http://www.omg.org/uml/.

[8] Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*, Addison Wesley Longman, Inc., USA, 1998.

[9] Schmuller J., *Sams Teach Yourself UML in 24 Hours*, USA, Sams Publishing, 1999.

[10] Rufai, Raimi Ayinde, *New structural similarity metrics for UML models*, Master Thesis, King Fahd University of Petroleum and Minerals, 2003.

[11] D'Souza, D. and Wills, A.C. Objects, *Components and Frameworks with UML: The Catalysis Approach,* Addison Wesley, Reading, MA, 1999.

[12] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, Addison Wesley Longman Ltd, 1998.

[13] Shumway M. F., Measuring Class Cohesion in Java, *Computer Science Department, Colorado State University,* June 11, 1997.

[14] Bieman J., Ott L., "Measuring Functional Cohesion", *IEEE Transactions on Software Engineering,* vol. 20, no. 8, August 1994.

[15] Bieman J. M., Kang B., "Measuring Design-Level Cohesion", *IEEE Trans. On Software Engineering*, vol. 24, no. 2, February 1998.

[16] Eder J., Kappel G., Schrefl M., *Coupling and Cohesion in Object-Oriented Systems*, Technical Report of University Klagenfurt, Institute of Computer Science, 1993.

[17] Chidamber S. R., Kemerer C. F.,"Towards a Metrics Suite for Object Oriented Design", in A. Paepcke, (ed.) *Proc. Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91), SIGPLAN Notices* vol.26, no.11, 197-211, 1991.

[18] Chidamber S., Kemerer C., "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, vol 20, no. 6, June 1994.

[19] Hitz M., Montazeri B., "Measuring Coupling and Cohesion in Object-Oriented systems", *in Proc. Int. Symposium on Applied Corporate Computing,* Monterrey, Mexico, October 1995.

[20] Hitz M., Montazeri B., "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective", *IEEE Transaction on Software Engineering*, vol. 22, no. 4, April 1996.

[21] Bieman J. M., Kang B., "Cohesion and Reuse in an Object-Oriented System", in *Proc. ACM Symp. Software Reusability (SSR'94),* pp 259-262, 1995.

[22] Henderson-Sellers B., Constantine L. L., Graham I. M., "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)", *Object Oriented Systems 3*, pp 143-158., 1996.

[23] Briand L., Morasca S., Basili V., "*Defining and Validating High-Level Design Metrics*", Computer Science Technical Report CS-TR 3301, University of Maryland at College Park, 1994.

[24] Bansiya J., Etzkorn L., Davis C., and Li W., "A Class Cohesion Metric for Object-Oriented Designs", *Journal of Object-Oriented Programming*, pp 47-52, January 1999.

[25] Chae H. S., Kwon Y. R., Bae D., "A cohesion Measure for Object-Oriented Classes", *Software-Practice & Experience*, vol.30, no.12, pp 1405-1431, October 2000.

[26] Jarallah Ghamdi, Wasiq M., and Ahmed M. A., "Principle and Metrics for Cohesion-Based Object-Oriented Component Assessment", *Confidential Draft Copy*, 2001.

[27] Aman H., Yamasaki K., Yamada H. Noda M., "A Proposal of Class Cohesion Metrics Using Sizes of Cohesive Parts", *T. welzer et al.(Eds.), Knowledge-based Software Engineering,* pp.102-107, IOS Press, Sept. 2002.

[28] Robert C. Martin, *Designing Object Oriented Applications using UML*, 2d. Ed, Prentice Hall, 1999.

[29] Article by Kirk Knoernschild titled "Java Package Functionality", August 2002.

[30] T. Vernazza, G. Granatella, G. Succi, L. Benedicenti, M. Mintchev "Defining Metrics for Software Components." *World Multiconference on Systemics, Cybernetics and Informatics,* vol. 11, pp. 16-23, Orlando, Florida, July 2000.

[31] Briand, L., S. Morasca, V. Basili, "Property-based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, 22(1), 1996.

[32] Gamma, E., R. Helm, R. Johnson, J. Vlissidies, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing, 1994.

[33] OMG{ XE "Object Management Group" } (Object Management Group) Specification, XMI{ XE "XMI" } (XML Metadata Interchange) Retrieved December 06, 2003 from the World Wide Web: http://www.omg.org/technology/documents/formal/xmi.htm.

[34] David Garlan and Mary Shaw, "An introduction to software architecture", In *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific Publishing Company, 1993.

[35] Paul Adamczyk, "The Anthology of the Finite State Machine Design Patterns", *The 10th Conference on Pattern Languages of Programs,* 2003.

[36] Lionel C. Briand, John W. Daly, and Jürgen Wüst,"A Unified Framework for Coupling Measurement in Object-Oriented Systems", *Fraunhofer Institute for Experimental Software Engineering,* Kaiserslautern, Germany. ISERN-96-14.

[37] Fenton N. E., "Software Measurement: A necessary scientific basis", *IEEE Trans. Software Eng.*, vol. 20, no. 3, pp. 199-206 March 1994.

[38] Hermadi, I., El-Badawi, K., and Al-Ghamdi, J., "Theoretical Validation of Cohesion Metrics in Object Oriented Systems". *International Arab Conference on Information Technology 2002 (ACIT2002)*, pp. 16-19, Dec. 2002.

[39] Sami Mäkelä and Ville Leppänen, "Taking Purpose of Class into Consideration in Cohesion Metrics", *Department of Information Technology, University of Turku and TUCS*, Finland.

[40] Briand et al., "*Theoretical and Empirical Validation of Software Product Measures*", International Software engineering research network technical report #ISERN-95-03, 1995.

[41] Nguyen, Munsen and John Boyland, “An Infrastructure for development of Object-Oriented, Multi-level Configuration Management Services”, International *conference of Software Engineering, St Louis, Missouri, U.S.A,* May 2005.

[42] J. Odell, H. Parunak, and B. Bauer, “Extending UML for agents”, *AgentOriented Information Systems Workshop at the 17th National conference on Artificial Intelligence,* 2000.

[43] David Grosser, A. Sahraoui and Petko Valtchev, "Analogy Based Software Quality Prediction", 7th Workshop on Quantitative Approaches in Object-OrientedSoftware Engineering, 2003

Vita

Syed Manzoor Hussain has been a Graduate (M.S) Student and Research Assistant at Information and Computer Science, King Fahd University of Petroleum and Minerals. He did Bachelor of Engineering (B.E) in Computer Science from Gulbarga University, India. He has a Journal publication to his credit titled “Mosix Evaluation on a Linux Cluster”. His research interest is in Software metrics. Currently he is working as an Applications developer in KFUPM, where he is involved in ERP using Oracle E-Business Suite. He can be reached at manzoor@ccse.kfupm.edu.sa.